



About this Book

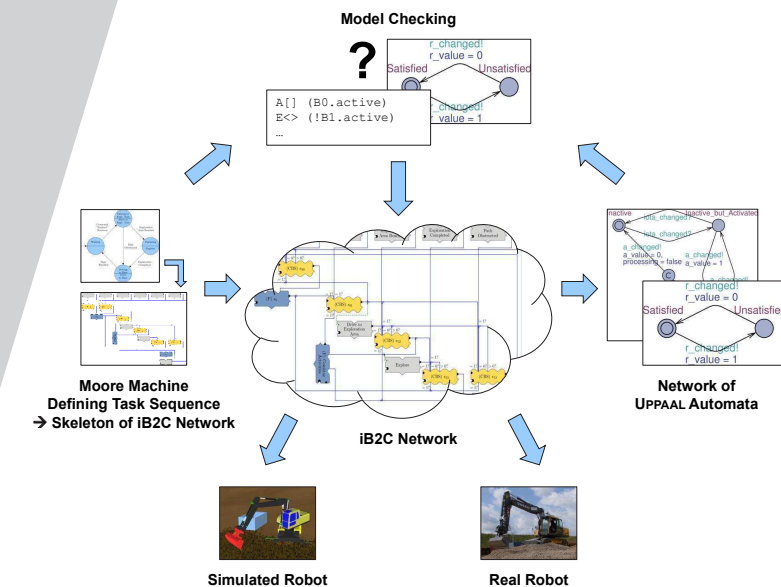
This dissertation introduces a novel integrated concept for the development and verification of behaviour-based systems that realise task sequences. The presented concept describes how sequences of tasks can be encoded in behaviour-based systems. Said sequences are defined as Moore machines, which are then automatically transferred into skeletons of behaviour networks.

These automatically created networks are typically extended and modified manually, which can easily lead to errors. Therefore, the concept also proposes an approach to verifying the correctness of the resulting networks. It is based on modelling behaviour-based systems as networks of automata, which are given as input to the model checking toolbox Uppaal in order to verify crucial requirements. The soundness of the presented concept is shown using the control system of an autonomous bucket excavator.

About the Author

From 2001 to 2007, Christopher Armbrust studied computer science with a focus on robotics at the University of Kaiserslautern, Germany. He received his Diploma in 2007. Since then, he has been pursuing his PhD at the university's Robotics Research Lab. His research interests include behaviour-based control systems and their application to autonomous mobile (in particular off-road) robots.

Design and Verification of Behaviour-Based Systems



Christopher Armbrust

Design and Verification of Behaviour-Based Systems Realising Task Sequences

Design and Verification of Behaviour-Based Systems Realising Task Sequences

Christopher Armbrust

Vom Fachbereich Informatik der
Technischen Universität Kaiserslautern

zur Verleihung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN 978-3-8439-2261-6

Zur Begutachtung eingereicht am:	24. November 2014
Datum der wissens. Aussprache:	3. Juli 2015
Vorsitzender:	Prof. Dr. Reinhard Gotzhein
Erster Berichterstatter:	Prof. Dr. Karsten Berns
Zweiter Berichterstatter:	Prof. Dr. Roland Meyer
Dekan:	Prof. Dr. Klaus Schneider
Zeichen der TU im Bibliotheksverkehr:	D 386

© Verlag Dr. Hut, München 2015
Sternstr. 18, 80538 München
Tel.: 089/66060798
www.dr.hut-verlag.de

Die Informationen in diesem Buch wurden mit großer Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und ggf. Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene fehlerhafte Angaben und deren Folgen.

Alle Rechte, auch die des auszugsweisen Nachdrucks, der Vervielfältigung und Verbreitung in besonderen Verfahren wie fotomechanischer Nachdruck, Fotokopie, Mikrokopie, elektronische Datenaufzeichnung einschließlich Speicherung und Übertragung auf weitere Datenträger sowie Übersetzung in andere Sprachen, behält sich der Autor vor.

1. Auflage 2015

To Lena

Preface

Writing a dissertation about the design and the verification of behaviour-based systems has been a complex task and I would like to take the opportunity to thank the people who supported me. First of all, my thanks go to Prof. Dr. Karsten Berns who made it possible that I could pursue my research in his group. He let me choose a scientific area fitting my interests and granted me a lot of freedom during my studies. Especially during the final work on my dissertation, he helped me with valuable suggestions on how to present my scientific contributions. Moreover, I am grateful for the support of Prof. Dr. Roland Meyer, who was the second supervisor of my dissertation. He guided me during my first steps in the area of verification and answered many questions on the use of model checking. Furthermore, I also thank Prof. Dr. Reinhard Gotzhein for chairing the defence of my dissertation.

I very much enjoyed working at the Robotics Research Lab due to the pleasant atmosphere created by my (former and present) colleagues. My thanks go to every one of them. In particular, I would like to mention the former members of the RAVON team, with whom I spent countless hours trying to make an off-road robot navigate autonomously in unknown environments: Tim Braun, Tobias Föhst, Bernd-Helge Leroch, Martin Proetzsch, Alexander Renner, and Gregor Zolynski. I especially enjoyed the close and intense collaboration during the ELROB trials. Moreover, I would like to thank Atabak Nejadfard, Thomas Pfister, Sascha Steffens, and Massimo Tosa for their hard work in the ICARUS project. Their great commitment significantly contributed to making the project a success. I am also grateful for the support of the lab's secretary Rita Broschart, our former technician Lothar Gauß, and his successor Sascha Steffens. Special thanks go to my colleagues Thorsten Ropertz and Lisa Kiebusch, with whom I have conducted extensive research on the verification of behaviour-based systems.

Furthermore, I am grateful for the work of my former students Maryla Rittmann, Stephan Rohr, and Thorsten Ropertz. Their technical work helped me prove the soundness of several concepts of my thesis.

My colleagues Lisa Kiebusch, Max Reichardt, and Daniel Schmidt as well as my cousin Marion have proofread parts of my dissertation. I highly appreciated their feedback, which helped me improve this document.

Moreover, I owe thanks to the team of the SCI (Service-Center Informatik) at the University of Kaiserslautern for providing a powerful computation server on which I could execute long-running verification processes.

Last but not least, I would like to thank my family, in particular my parents and my life partner Lena, for their constant support.

Abstract

Since their invention in the 1980s, behaviour-based systems have become very popular among roboticists. Their component-based nature facilitates the distributed implementation of systems, fosters reuse, and allows for early testing and integration. However, the distributed approach necessitates the interconnection of many components into a network in order to realise complex functionalities. This network is crucial to the correct operation of the robotic system. There are few sound design techniques for behaviour networks, especially if the systems shall realise task sequences. Therefore, the quality of the resulting behaviour-based systems is often highly dependant on the experience of their developers.

This dissertation presents a novel integrated concept for the design and verification of behaviour-based systems that realise task sequences. Part of this concept is a technique for encoding task sequences in behaviour networks. Furthermore, the concept provides guidance to developers of such networks. Based on a thorough analysis of methods for defining sequences, Moore machines have been selected for representing complex tasks. With the help of the structured workflow proposed in this work and the developed accompanying tool support, Moore machines defining task sequences can be transferred automatically into corresponding behaviour networks, resulting in less work for the developer and a lower risk of failure.

Due to the common integration of automatically and manually created behaviour-based components, a formal analysis of the final behaviour network is reasonable. For this purpose, the dissertation at hand presents two verification techniques and justifies the selection of model checking. A novel concept for applying model checking to behaviour-based systems is proposed according to which behaviour networks are modelled as synchronised automata. Based on such automata, properties of behaviour networks that realise task sequences can be verified or falsified. Extensive graphical tool support has been developed in order to assist the developer during the verification process.

Several examples are provided in order to illustrate the soundness of the presented design and verification techniques. The applicability of the integrated overall concept to real-world tasks is demonstrated using the control system of an autonomous bucket excavator. It can be shown that the proposed design concept is suitable for developing complex sophisticated behaviour networks and that the presented verification technique allows for verifying real-world behaviour-based systems.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	2
1.3	Outline	4
2	Sequences	7
2.1	Representation of Sequences	9
2.1.1	Finite-State Machines	9
2.1.2	UML Diagrams	14
2.1.3	Petri Nets	16
2.2	Realisation of Sequences	20
2.2.1	Centralised Realisation of Sequences in Behaviour-Based Systems .	25
2.2.2	Decentralised Realisation of Sequences in Behaviour-Based Systems	32
2.2.2.1	Decentralised Realisation of Sequences Without Using Special Inter-Behaviour Connections	32
2.2.2.2	Decentralised Realisation of Sequences Using Special Inter-Behaviour Connections	35
2.3	Discussion	40
3	Encoding Task Sequences in iB2C Networks	43
3.1	Realising Behaviour Activity Sequences in iB2C Networks	44
3.1.1	Local Coordination Behaviour	44
3.1.2	Example Applications	57
3.1.2.1	Turning Manoeuvre	59
3.1.2.2	Dead End Detection	66
3.1.3	Discussion	71
3.2	Transferring Moore Automata into iB2C Networks	72
3.2.1	Transformation Process	72
3.2.2	Example Application: Exploration Task	78
3.2.3	Complexity	83
3.2.4	Graphical Tool Support	86
3.2.5	Discussion	89

4	Verifying iB2C Networks	93
4.1	Verification Techniques	94
4.1.1	Deductive Reasoning	95
4.1.2	Model Checking	96
4.1.2.1	Representations of Kripke Structures	99
4.1.2.2	The Model Checking Toolbox Uppaal	101
4.2	Modelling iB2C Behaviour Networks as Networks of Uppaal Automata . .	106
4.2.1	Mapping Standard iB2C Behaviours to Uppaal Automata	108
4.2.2	Mapping iB2C Behaviour Networks to Networks of Uppaal Automata	114
4.2.3	Mapping iB2C Fusion Behaviours to Uppaal Automata	116
4.2.3.1	Modelling the Activity Calculation of a Fusion Behaviour With a Single Automaton	117
4.2.3.2	Modelling the Activity Calculation of a Fusion Behaviour With Multiple Automata	118
4.2.3.3	Modelling the Target Rating Calculation of a Fusion Be- haviour With a Single Automaton	121
4.2.4	Mapping iB2C CBS Behaviours to Uppaal Automata	123
4.2.5	Quantitative Aspects	132
4.3	Verifying iB2C Networks Using Model Checking	138
4.3.1	Example Application: Navigation System of RAVON	139
4.3.2	Tool-Assisted Verification of Behaviour Networks	147
4.3.2.1	Properties of Behaviour-Based Systems	148
4.3.2.2	Graphical Query Design Using Query Graphs	153
4.3.2.3	Generation of Queries from Query Graphs	154
4.3.2.4	Visualisation of Traces	156
4.4	Discussion	156
5	Application Example	161
5.1	Designing an iB2C Network Realising an Excavation Task	161
5.2	Verifying an iB2C Network Realising an Excavation Task	167
5.2.1	Verifying Predefined Properties	167
5.2.2	Analysing Partially Unknown Systems	173
5.3	Discussion	180
6	Conclusion	181
6.1	Summary	181
6.2	Evaluation	185
6.3	Outlook on Future Work	185
	Appendices	187
A	Robot Control Architectures	189
A.1	Deliberative Robot Control Architectures	190
A.2	Reactive Robot Control Architectures	191
A.3	Hybrid Robot Control Architectures	193

B MCA and FINROC	199
B.1 MCA	199
B.2 FINROC	202
Bibliography	205
Index	217

1. Introduction

1.1 Motivation

The first commercial robots were industrial robots, machines that repeatedly fulfil simple tasks in a faster or more precise way than a human would be able to. They usually operate enclosed in a work cell, which limits their world to a well-structured environment with strictly controlled changes. Hence, industrial robots are normally not equipped with sophisticated sensor systems, but mainly with internal sensors for measuring their pose. As a consequence, they typically feature simple control systems realising a single control loop.

In recent years, technologies from the area of robotics have more and more found their way into other areas: Obstacle detection sensors enable cars to warn their drivers of nearby obstacles, while image processing systems allow them to point out road signs. Based on sophisticated localisation and navigation techniques, heavy agricultural machines are able to (semi-)autonomously harvest fields. With the help of complex 3D sensors and sophisticated data processing algorithms, construction vehicles of the future shall autonomously operate on construction sites with only limited human intervention. All these robotic systems operate in complex, changing outdoor environments. They share those environments with humans, with whom they sometimes even have to interact. This necessitates more sophisticated control systems that are able to fulfil versatile, complex tasks.

To satisfy the needs of (future) autonomous outdoor vehicles, their control systems must allow for fast reactions to changes in the vehicles' environments. In the 1980s, a new type of robot control architectures was invented: the behaviour-based architectures. Contrary to the previously existing control architectures that implement the sense-plan-act loop, behaviour-based architectures aim at a tight coupling between sensors and actors. In contrast to classic architectures, in which few large components take care of the robot's task, behaviour-based architectures feature a highly modularised topology in which several behaviours interact. This yields a higher reusability and facilitates distributed implementation as well as early testing and integration. These features make

behaviour-based architectures well-suited for realising tasks like collision avoidance on, e.g., construction vehicles.

A downside of the behaviour-based approach is that the interaction of the single behaviours in a network is crucial to the correct operation of the complete system. This interaction is typically complex and often necessitates a huge number of inter-behaviour communication links. Behaviour-based systems are often created in a manual, intuitive way due to the lack of sound modelling techniques for systems executing task sequences and developers experienced only in the classic way of structuring systems. The result is a high risk of introducing errors. Furthermore, the high degree of distribution of functionality makes the detection of errors and the identification of their causes difficult.

The work at hand targets these drawbacks and provides support for the design of behaviour-based systems that realise sequences of tasks as well as for the verification of such systems.

1.2 Objectives

Figure 1.1 illustrates the overall concept underlying the work at hand. The concept is centred around a behaviour network whose design and verification it describes.

The design of the network starts with the definition of a complex task, which is a task that consists of a sequence of subtasks. Task sequences often occur in the application fields of sophisticated robots. A typical example is the exploration of an unknown environment, which usually consists of a robot driving from its base to an unknown area, exploring the area, and returning to the base after the completion of the exploration. This task is highly relevant in fields like search-and-rescue or space robotics.

Based on the definition of a task sequence, a behaviour network shall be created that realises this task sequence, i.e. that takes care of executing all tasks in the correct order. The creation of the behaviour network shall be done according to a precisely defined technique that facilitates the work of the developer and reduces the risk of failure. Furthermore, the developer shall receive support through the provision of sophisticated tools as well as a high degree of automation. The resulting network can be used to control a simulated robot or be deployed on a real machine. In Fig. 1.1, this is illustrated with a simulated and a real excavator.

The behaviour-based system designed in the described way realises the defined sequence of tasks. However, it is not a complete robot control system, but a high-level component that has to be integrated with other components into a larger system that can then be employed on a robot system. Furthermore, additional components may be added (manually) after the development of the original system has been completed.

Both steps—the integration into a larger system as well as the extension with further components—entail the risk of introducing errors into the system. These errors can be the cause of an incorrect execution of the defined task sequence by the behaviour network. Thorough testing can help in reducing the number of errors, but it cannot prove their total absence. Therefore, verification is necessary to ensure the correct operation of the final system. For this purpose, a formal model of the system shall be created and verified against the specification. Again, powerful tool support and a high degree of automation shall be available in order to facilitate the work of the developer.

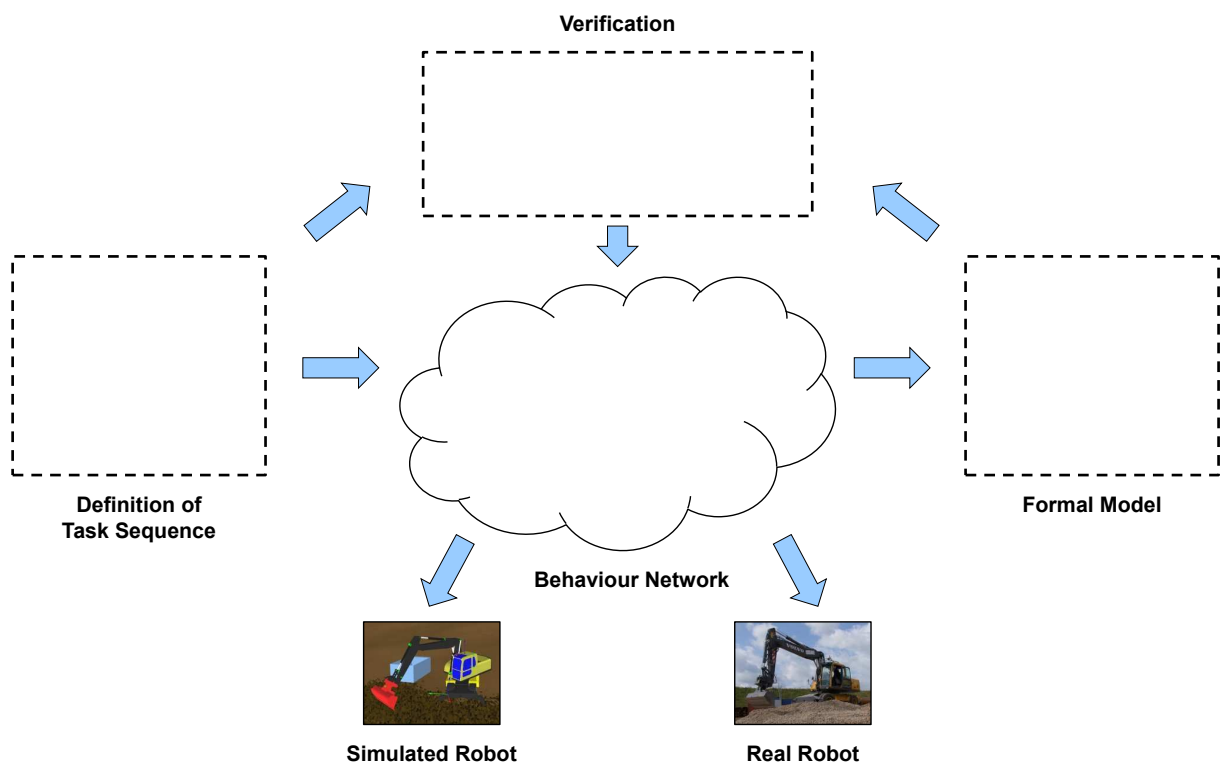


Figure 1.1: The concept of this doctoral thesis: a behaviour network controls a real robot or its simulated counterpart. Here, this robot is THOR, a construction vehicle (see Sec. 5.1). The behaviour network takes care of the execution of a task sequence. A formal model of the network and the task description are used for verifying the correct operation of the network.

The above considerations lead to the following two objectives of the thesis at hand:

Objective 1: Development of a Technique for Realising Task Sequences in Behaviour Networks

The purpose of developing a technique for realising task sequences in behaviour networks is to provide assistance during the design of behaviour-based robot control systems which realise sequences of tasks. The technique shall provide information about how to transfer an application-near description of a complex task consisting of a sequence of subtasks into a well-structured behaviour network that executes the specified task.

Objective 2: Development of a Technique for Verifying Behaviour Networks that Realise Task Sequences

The purpose of developing a technique for verifying behaviour networks that realise task sequences is to provide assistance during the verification of behaviour-based robot control systems which realise sequences of tasks. The technique shall take into account the distributed nature of behaviour-based systems as well as the importance of the behaviour interconnections and allow for verifying complex behaviour networks.

Objective 1 targets the design aspect of this thesis, which is illustrated by the dashed rectangle on the left hand side of Fig. 1.1. The verification aspect of this dissertation is the target of Objective 2, which is illustrated by the dashed rectangles on the right hand side (creation of formal model) and on the top (actual verification) of Fig. 1.1, respectively. In the course of this dissertation, the empty boxes in Fig. 1.1 will be filled according to the presented results.

1.3 Outline

The remainder of this thesis is structured as follows.

The purpose of Chap. 2 is to provide fundamental information about sequences that is relevant to this dissertation. Its first part will give an overview of different methods for representing sequences. Such methods are necessary for specifying complex systems that realise sequences of tasks. The method of choice for this thesis will be identified and the selection of this very method will be justified.

Furthermore, the second part of Chap. 2 will show how sequences of tasks can be realised in robot control systems. It will be explained that the realisation depends heavily on the architecture of the control system. The advantages and drawbacks of behaviour-based systems will be discussed. Based on this discussion, behaviour-based approaches will be selected for the work described in this dissertation. The selection of a suitable method for implementing sequences in behaviour-based systems will also be presented in Chap. 2. In addition, several design decisions will be formulated as guidelines for the remainder of this thesis.

In Chap. 3, a technique for realising complex sequences in a behaviour-based system will be introduced. This technique will not only allow for designing behaviour networks that can sequentially execute actions, but it will also render possible the creation of behaviour

networks that detect the sequential occurrence of structures in a robot's environment. With this technique, the technical prerequisite for realising task sequences in behaviour networks will be available.

Chapter 3 will also present a concept for designing a behaviour-based system that executes a complex task sequence. The proposed approach will allow for defining a system graphically in an application-oriented way and automatically creating the corresponding behaviour network afterwards. Thus, it will be possible to easily realise complex task sequences using behaviour-based systems without the need to manually connect behaviours.

The distributed nature of behaviour-based systems makes their verification difficult. This in particular applies to the complex networks which have been created according to the concepts that will be presented in Chap. 3. Nevertheless, said networks should be verified in case they are integrated into a larger network or extended with other components (see Sec. 1.2). Therefore, a verification technique based on the interaction of behaviours has been developed and will be described in Chap. 4. Based on a brief overview of different approaches to verification, the choice of the verification technique will be justified. The technique consists of two steps: the creation of a formal model of the system to verify and the actual verification process. Both steps will be described in detail in Chap. 4. The implementation of the proposed technique will support the developer by providing a graphical user interface and several automatisms. Further design decisions will be formulated in order to guide the work on the verification concept.

In Chap. 5, an example of the application of the concepts presented in the preceding chapters will be given. It is based on the sequence of tasks an autonomous bucket excavator has to execute in order to perform an excavation process. This example has been chosen as it demonstrates the applicability of the presented concepts to a real-world robot system. It will cover the complete overall concept of this thesis, ranging from the specification of the excavation task in an application-near way over the creation of the corresponding behaviour network to the verification of the correct operation of the resulting system.

Finally, the contents of this work will be summarised in Chap. 6. The major results will be listed and the implementation of the previously made design decisions will be described. Furthermore, the scientific achievements of this dissertation will be evaluated and options for improvements will be identified. Subsequently, an outlook on future work will be given.

Throughout this thesis, the presented concepts are illustrated by means of examples. These are based on three robots: an exploration robot (see Chap. 2), the autonomous off-road research platform RAVON (see Sec. 3.1.2), and finally the autonomous bucket excavator THOR (see Fig. 1.1 and Sec. 5.1) as an example of a construction vehicle.

2. Sequences

The term “sequence” is important for the work at hand as the ability to execute a sequence of tasks distinguishes a high-level system that is able to fulfil complex tasks from a low-level system that can only deal with simple problems. Hence, it shall be defined formally here. The following definitions of infinite sequences and finite sequences are based on those given in [Ledermann 82] and [Bronšteín 08]:

Definition 2.1: Infinite Sequence

An *infinite sequence* of real numbers is a function $s : \mathbb{N} \rightarrow \mathbb{R}$. The elements of s are $s(0), \dots, s(n-1), \dots$ or simply $s_0, \dots, s_{n-1}, \dots$, with $s(i) = s_i$ being called the i th term of the sequence. The notation $\{s_k\}$ with $k = 1, 2, \dots$ is also common.

Definition 2.2: Finite Sequence

A *finite sequence* of real numbers is a function $s : \{0, \dots, n-1\} \rightarrow \mathbb{R}$. The elements of s are $s(0), \dots, s(n-1)$ or simply s_0, \dots, s_{n-1} , with $s(i) = s_i$ being called the i th term of the sequence. The notation $\{s_k\}$ with $k = 1, \dots, n-1$ is also common.

These definitions of sequences are too abstract for discussing the sequential execution of tasks. Hence, the term “task sequence” shall be defined based on Defs. 2.1 and 2.2:

Definition 2.3: Task Sequence

Be \mathbb{T} a set of tasks. Then a (finite) *task sequence* is a function $s : \{0, \dots, n-1\} \rightarrow \mathbb{T} \times \mathbb{R}$ with elements $s(i) = s_i$ defined as $s_0 = (T_{j_0}, t_0), \dots, s_{n-1} = (T_{j_{n-1}}, t_{n-1})$ for which the following holds:

1. i is an index for the elements of s , i.e. $0 \leq i \leq n-1$.
2. j_i are indices into the set of tasks \mathbb{T} , i.e. $0 \leq j_i \leq |\mathbb{T}| - 1$.
3. $t_i \in [0, T] \subset \mathbb{R}$ denote distinct points in a time interval ranging from 0 to T with $i < k \implies t_i < t_k$.
4. w indicates whether the robot is working on a task, i.e. $w : \mathbb{T} \times \mathbb{R} \rightarrow \{0, 1\}$ with $w(T, t) = \begin{cases} 1 & \text{if the robot is working on task } T \text{ at time } t \\ 0 & \text{else} \end{cases}$.
5. The robot starts working on task T_{j_i} at t_i , i.e. $\forall s_i = (T_{j_i}, t_i) : (\exists \delta > 0 : \forall \varepsilon \text{ with } 0 < \varepsilon \leq \delta : w(T_{j_i}, t_i - \varepsilon) = 0) \wedge (w(T_{j_i}, t_i) = 1)$.

The example of an exploration robot shall be used to illustrate this definition. An autonomous mobile robot (like, for example, RAVON, depicted in Fig. 2.20) shall navigate to an unexplored area, execute an exploration task there, and then return to its base. In order to realise this sequence of tasks, the control system of the robot shall be able to execute three subtasks: *Drive to Exploration Area*, *Explore*, and *Drive to Base*. This example will be used in the remainder of this thesis several times for illustrating different ways of how to define sequences. In this example, $n = 3$ and $\mathbb{T} = (T_0, T_1, T_2)$ with $T_0 = \text{Drive to Exploration Area}$, $T_1 = \text{Explore}$, and $T_2 = \text{Drive to Base}$. As a result, the sequence s representing the given tasks in the correct order is defined as follows:

$$\begin{aligned} s_0 &= (T_0, t_0) = (\text{Drive to Exploration Area}, t_0) \\ s_1 &= (T_1, t_1) = (\text{Explore}, t_1) \\ s_2 &= (T_2, t_2) = (\text{Drive to Base}, t_2) \end{aligned}$$

In many cases, the exact time t_i at which the robot starts working on a task T_{j_i} is insignificant. What is important is that $i < k \implies t_i < t_k$, i.e. that the points at which the robot starts working on the tasks are ordered in a sequence. Therefore, the sequence of the example can be written in a shortened form:

$$\begin{aligned} s_0 &= T_0 = \text{Drive to Exploration Area} \\ s_1 &= T_1 = \text{Explore} \\ s_2 &= T_2 = \text{Drive to Base} \end{aligned}$$

In this chapter, different ways of representing task sequences will be presented (see Sec. 2.1). Furthermore, it will be shown how sequences can be realised in robot control systems (see Sec. 2.2).

2.1 Representation of Sequences

This section presents several methods for modelling sequences of (sub)tasks. Modelling sequences is important during the development of sophisticated robot control systems as complex tasks are built up of sequences of subtasks. Hence, the techniques applied during the development process should allow for representing sequences in a concise way.

2.1.1 Finite-State Machines

A common way of representing sequences is the use of state machines, sometimes also called state automata. There are many different types of state machines and thus, there is a lot of literature dealing with automata theory (see, e.g. [Shields 89], [Wagner 06], and [Sakarovitch 09]). However, the used terms and definitions vary. Some authors, for example, use “state machine” and “state automaton” interchangeably, others distinguish between the two. In this thesis, the two terms are *not* distinguished. The definitions of automata that can be found in the literature partly even differ in the number of elements an automaton consists of. For the work at hand, an extensive description of the different types of automata is not necessary. Instead, only brief definitions of the automata needed as basis for this work are given.

State machines can be roughly classified into two categories: acceptor automata (or simply acceptors) and transducer automata (or simply transducers). Acceptors read inputs, transition between states depending on the inputs, and finally transition into a terminal state, which indicates whether the automaton has accepted the input (hence the name acceptor). An acceptor can be used to define a language, meaning that the language consists of all the words that the automaton accepts as input. Below is a formal definition of an acceptor:

Definition 2.4: Acceptor

An *acceptor* is a 5-tuple (S, s_I, Σ, T, F) , with S being a set of states, $s_I \in S$ an initial state, Σ an input alphabet, $T : S \times \Sigma \rightarrow S$ a transition function, and $F \in S$ a set of final states.

In contrast to acceptors, transducers generate output. Transducers in which the output only depends on the current state are called “Moore machines”, while transducers in which the output depends on the current state and the input are called “Mealy Machines”. Below are formal definitions of these two types of transducers.

Definition 2.5: Moore Machine

A *Moore machine* is a 6-tuple $(S, s_I, \Sigma, \Lambda, T, G)$, with S being a set of states, $s_I \in S$ an initial state, Σ an input alphabet, Λ an output alphabet, $T : S \times \Sigma \rightarrow S$ a transition function, and $G : S \rightarrow \Lambda$ an output function.

Definition 2.6: Mealy Machine

A *Mealy machine* is a 6-tuple $(S, s_I, \Sigma, \Lambda, T, G)$, with S being a set of states, $s_I \in S$ an initial state, Σ an input alphabet, Λ an output alphabet, $T : S \times \Sigma \rightarrow S$ a transition function, and $G : S \times \Sigma \rightarrow \Lambda$ an output function.

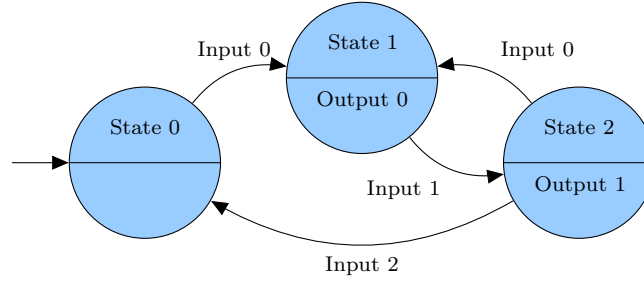


Figure 2.1: A state diagram visualising an example Moore machine.

T and G may be partial functions, i.e. there may be states in which certain inputs do not trigger a transition and there may be states in which there is no output. As can be seen, the only difference in the formal definitions of Moore and Mealy machines is the definition of the output function G . All sets that appear in the above definitions are finite, hence these automata are also called *finite-state automata* (FSA) or *finite-state machines* (FSM).

State machines are often represented graphically using state diagrams. These consist of circles and interconnecting arrows along with labels. Circles are used to represent states, while an arrow between two circles indicates that there is a transition between the states represented by the two circles. An initial state is marked with an arrow going to the corresponding circle whose origin is not connected to any circle. Final states are marked with double lines. The name of a state is written within the corresponding circle. Inputs necessary for transitioning from one state to another are written as labels to the corresponding arrow. In case of a Mealy machine, outputs are also written to edges, separated from the input with a forward slash. For Moore machines, outputs are written within the corresponding circle below the name of the state.

Figure 2.1 depicts an example of a state diagram that represents the Moore machine $(S, s_I, \Sigma, \Lambda, T, G)$ defined as follows:

$$\begin{aligned}
 S &= \{\text{State 0, State 1, State 2}\} & s_I &= \text{State 0} \\
 \Sigma &= \{\text{Input 0, Input 1, Input 2}\} & \Lambda &= \{\text{Output 0, Output 1}\} \\
 T : T(\text{State 0, Input 0}) &= \text{State 1} & G : G(\text{State 0}) &= \varepsilon \\
 T(\text{State 1, Input 1}) &= \text{State 2} & G(\text{State 1}) &= \text{Output 0} \\
 T(\text{State 2, Input 0}) &= \text{State 1} & G(\text{State 2}) &= \text{Output 1} \\
 T(\text{State 2, Input 2}) &= \text{State 0}
 \end{aligned}$$

As can be seen from the example, FSMs can be used to represent non-linear sequences, i.e. the conditional sequences described in [Gat 94] (see Sec. 2.2) could also be specified using FSMs.

When modelling task sequences using finite-state automata, the first question to answer is what the individual elements of the automata shall represent. One approach is to let each state of an automaton correspond to a certain subtask that the robot is about to perform. A transition between two states then represents the change of the subtask that the robot

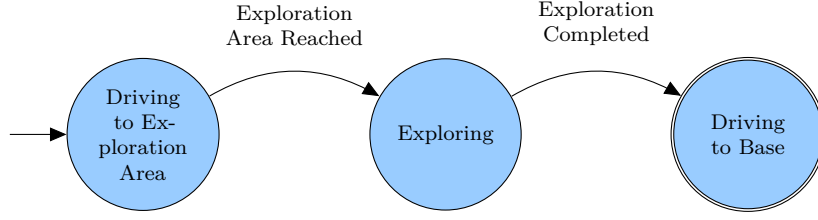


Figure 2.2: A state diagram visualising an acceptor automaton that realises the example of a robot exploring an area.

is currently working on. In other words, a transition between two states represents that the robot stops working on one subtask and starts working on another one. Typically, the input alphabet corresponds to certain conditions that have to be fulfilled before the robot is able to fulfil a certain task. The final states of an acceptor automaton can be used to represent that the work on a task has been completed (successfully or unsuccessfully). Finally, the output alphabet of a transducer automaton can be mapped to actions that the robot performs while working on a certain subtask.

In the following, several ways of modelling task sequences as finite-state automata are given. They will be illustrated using the example of an exploration robot introduced above. An acceptor representing the exploration task can be defined as follows:

$$\begin{aligned}
 S &= \{\text{Driving to Exploration Area, Exploring, Driving to Base}\} & F &= \{\text{Driving to Base}\} \\
 s_I &= \text{Driving to Exploration Area} \\
 \Sigma &= \{\text{Exploration Area Reached, Exploration Completed}\}
 \end{aligned}$$

$$\begin{aligned}
 T : T(\text{Driving to Exploration Area, Exploration Area Reached}) &= \text{Exploring} \\
 T(\text{Exploring, Exploration Completed}) &= \text{Driving to Base}
 \end{aligned}$$

This will result in the state diagram depicted in Fig. 2.2.

In a similar way, a Moore machine representing the sequence of subtasks can be defined:

$$\begin{aligned}
 S &= \{\text{Driving to Exploration Area, Exploring, Driving to Base}\} \\
 s_I &= \text{Driving to Exploration Area} \\
 \Sigma &= \{\text{Exploration Area Reached, Exploration Completed}\} \\
 \Lambda &= \{\text{Drive to Exploration Area, Explore, Drive to Base}\}
 \end{aligned}$$

$$\begin{aligned}
 T : T(\text{Driving to Exploration Area, Exploration Area Reached}) &= \text{Exploring} \\
 T(\text{Exploring, Exploration Completed}) &= \text{Driving to Base}
 \end{aligned}$$

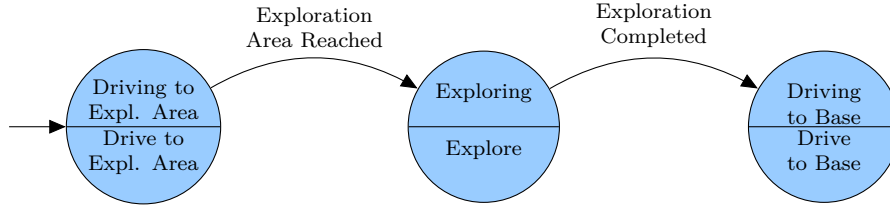


Figure 2.3: A state diagram visualising a Moore machine that realises the example of a robot exploring an area.

$$\begin{aligned}
 G : G(\text{Driving to Exploration Area}) &= \text{Drive to Exploration Area} \\
 G(\text{Exploring}) &= \text{Explore} \\
 G(\text{Driving to Base}) &= \text{Drive to Base}
 \end{aligned}$$

Figure 2.3 depicts the corresponding state diagram.

In this example, a state and the corresponding output are similar. However, it is reasonable to distinguish between the two. For example, while the robot may be in the state of driving to its base (i.e. the control system has decided to guide the vehicle to its base), this does not necessarily mean that the outputs of the vehicle (i.e. its actions) are in accordance with this. Under certain circumstances, an influence outside of the subsystem realising the exploration task might inhibit certain outputs. The realisation of a Moore machine as behaviour network presented in Sec. 3.2 makes use of this distinction by creating for each state a behaviour representing that state and a separate behaviour creating the output of that state.

Finally, here is the definition of a Mealy machine representing the exploration task. Another state (Waiting) and another input (Command “Explore!” Received) have been added to allow for the output “Drive to Exploration Area” to also appear in the automaton.

$$\begin{aligned}
 S &= \{\text{Waiting, Driving to Exploration Area, Exploring, Driving to Base}\} \\
 s_I &= \text{Waiting} \\
 \Sigma &= \{\text{Command “Explore!” Received, Exploration Area Reached,} \\
 &\quad \text{Exploration Completed}\} \\
 \Lambda &= \{\text{Drive to Exploration Area, Explore, Drive to Base}\}
 \end{aligned}$$

$$\begin{aligned}
 T : T(\text{Waiting, Command “Explore!” Received}) &= \text{Driving to Exploration Area} \\
 T(\text{Driving to Exploration Area, Exploration Area Reached}) &= \text{Exploring} \\
 T(\text{Exploring, Exploration Completed}) &= \text{Driving to Base} \\
 G : G(\text{Waiting, Command “Explore!” Received}) &= \text{Drive to Exploration Area} \\
 G(\text{Driving to Exploration Area, Exploration Area Reached}) &= \text{Explore} \\
 G(\text{Exploring, Exploration Completed}) &= \text{Drive to Base}
 \end{aligned}$$

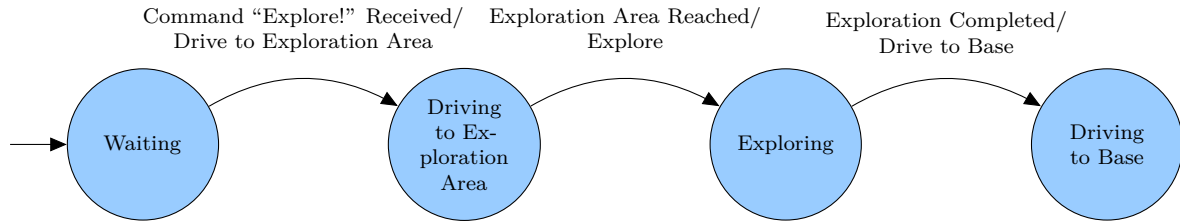


Figure 2.4: A state diagram visualising a Mealy machine that realises the example of a robot exploring an area.

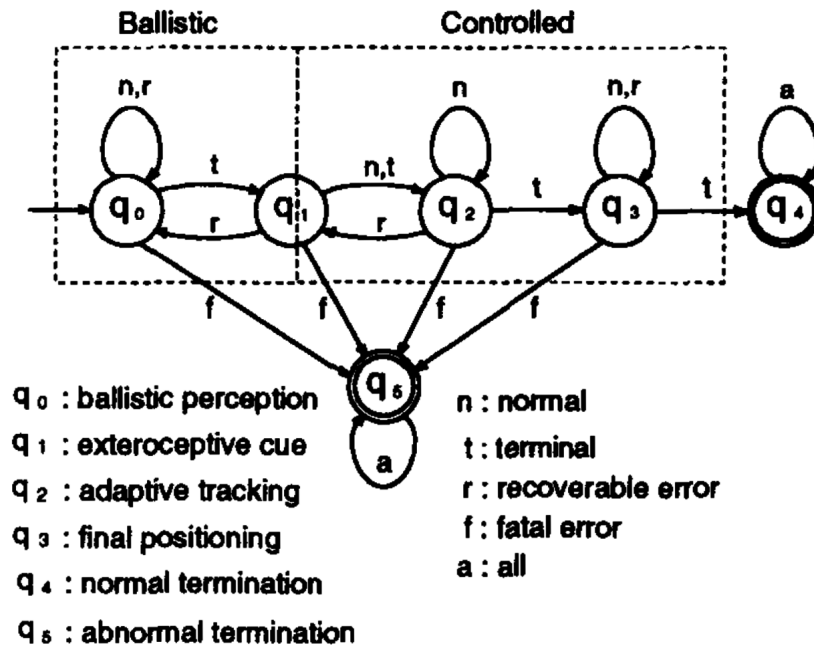


Figure 2.5: An acceptor automaton modelling the different steps of a docking operation. The robot uses different perception techniques (with corresponding states q_0 to q_3) for the different phases of the docking process (source: [Arkin 94]).

The use of state automata for defining task sequences is common. Hence, there are numerous publications describing the application of state automata in robotics. In [Arkin 98], for example, acceptor automata are used for representing sequences. The definition of acceptors given there has been developed in [Arkin 94] and is based on [Arbib 81]. It is similar to Def. 2.4, but lacks the input alphabet. This is implicitly given in the definition of the transition function, which maps the current state and an input to the following state. Figure 2.5 depicts an acceptor presented in [Arkin 94]. It models a control system realising docking operations of a mobile robot in manufacturing environments. This and further examples given in the above-mentioned publications by Ronald C. Arkin demonstrate that finite-state automata offer a convenient way for specifying different types of tasks and the control systems implementing them.

There are good reasons for representing a complex task consisting of a sequence of subtasks as a finite-state machine. For developers of robot control systems, who are typically from the field of computer science, state machines are a well-known representation with which

they most likely already have experience. The people specifying the tasks a robot shall fulfil, however, may come from the field of application of the system, which need not be computer science. In cases like the autonomous bucket excavator THOR (see Sec. 5.1), it can be the construction sector. As state machines are a comparably simple way of representing sequences, they can also be understood in an intuitive way by technical people not from the area of computer science, hence serving as a basis for discussions between users and developers during the development phase of a control system. A technical advantage of state automata is that one automaton can represent alternative sequences (by having multiple transitions going out of one state) as well as repeated execution of (sub)sequences (by having transitions going back to an earlier state in a sequence). Transducers have the advantage over acceptors to be able to represent actions of the robot in terms of outputs. Of the two types of transducers, Moore machines support the view that the robot realises a certain subtask in a certain state. This can be advantageous when it comes to realising a complex task as a behaviour network (see Sec. 5.1).

2.1.2 UML Diagrams

Different ways of specifying properties of software systems are offered by the Unified Modeling Language (UML). As written in [Booch 05], the UML is “a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system”. Sophisticated robots definitely are software-intensive systems—especially if it comes to robots that shall fulfil complex tasks. Hence, it seems obvious to specify complex tasks consisting of sequences of subtasks using UML diagrams.

UML supports the illustration of state machines (and hence of sequences) using so-called state diagrams. Using these diagrams, a developer can easily model the dynamic aspects of a system. UML state diagrams are similar to the state diagrams presented in Sec. 2.1.1, but differ in several aspects. For example, two types of actions can be attributed to states: entry effects, which are executed when entering a state, and exit effects, which are executed on leaving a state. Advanced states can also contain do-activities to model ongoing activities executed in the state. Moreover, event triggers and guard conditions are attributed to edges. Both of them are directly written to edges, with guard conditions enclosed in square brackets. A transition is eligible to fire if the corresponding event occurs and the associated guard condition is fulfilled. In general terms, the definition of state machines in UML is less restrictive than the ones given above. For example, it is possible to combine aspects of Moore and Mealy machines in one diagram. Furthermore, there are a number of additional constructs that facilitate the work of a system developer, like different types of special substates. Figure 2.6 presents an example from [Garousi 11]. It depicts the control system of an AIBO robot that is able to play soccer.

Figure 2.7 depicts an example of how the exploration task introduced above can be modelled using a UML state diagram. The figure shows a combination of a Moore machine and a Mealy machine: Two actions are attributed to states as do-activities (like in Moore machines) and one action is attributed to a transition (like in Mealy machines). While the ability to use a combination of the two types of machines offers more freedom to the developer of a system, it can make understanding or automatically processing the resulting diagrams more difficult.

Another type of UML diagrams is suited for modelling sequences: so-called activity diagrams. Like state diagrams, activity diagrams are used to model the dynamic aspects of a system.

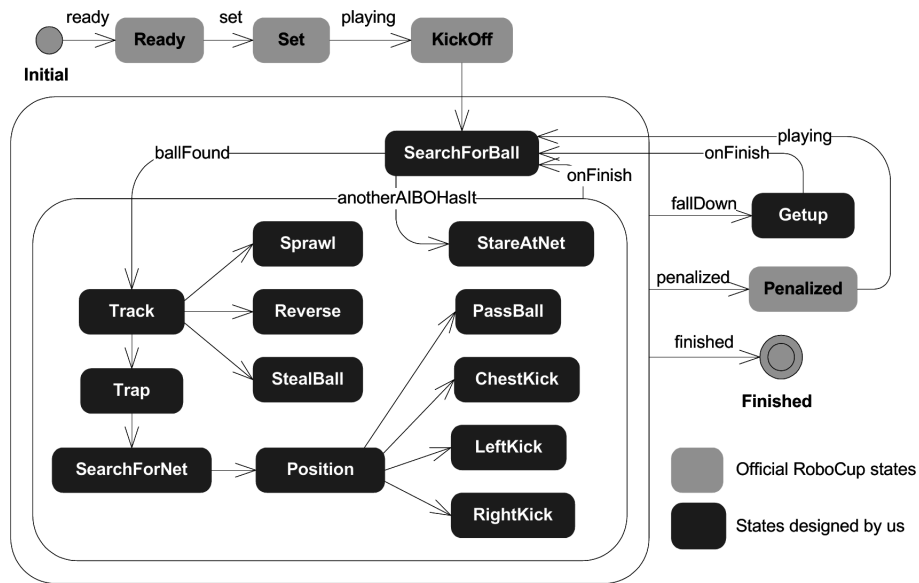


Figure 2.6: A UML state diagram depicting the control system of an AIBO robot that shall play soccer (source: [Garousi 11]).

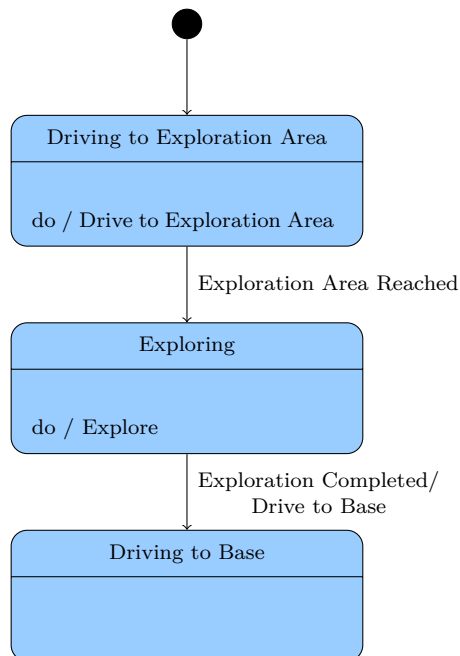


Figure 2.7: A UML state diagram visualising a state machine that realises the example of a robot exploring an area. It is a combination of a Moore machine and a Mealy machine: Two actions (outputs) are attributed to states as do-activities and one action is attributed to a transition.

Rounded boxes represent actions as well as activity nodes, which are groups of actions or other activity nodes. The control flow of a system is visualised using arrows from actions to their subsequent actions. This is sufficient for modelling linear sequences. For realising conditional sequences, points of branching and merging can be indicated using diamonds.

The authors of [Wongwirat 08] describe the use of UML for the development process of a mobile robot. They use different types of UML diagrams, among them activity diagrams for describing certain functions of the robot. Figure 2.8, for example, depicts the function that distinguishes different surfaces based on the value of a light sensor. The major weak point of the work is that the approach is only applied to a small LEGO robot operating in a very simple environment (see Fig. 2.9).

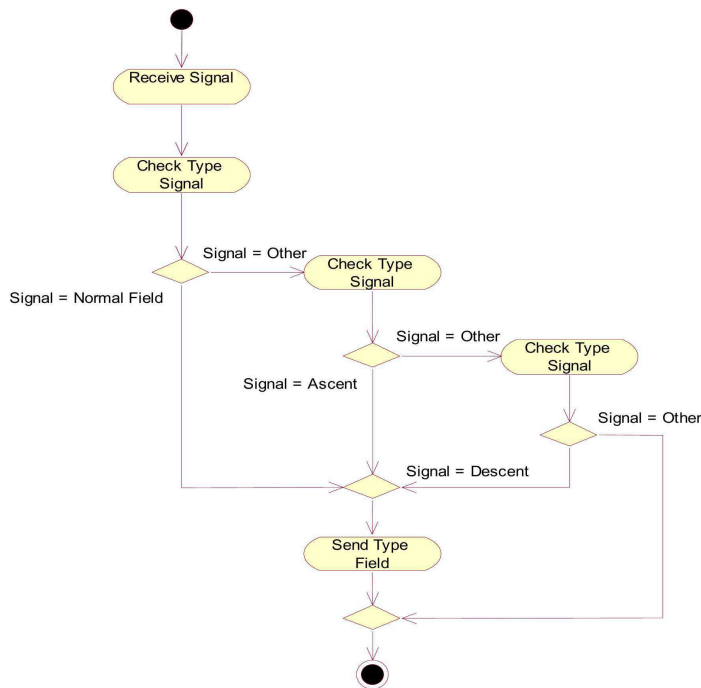


Figure 2.8: A UML activity diagram depicting a function that distinguishes different types of surfaces. It receives an input signal from a light sensor and outputs the type of surface (source: [Wongwirat 08]).

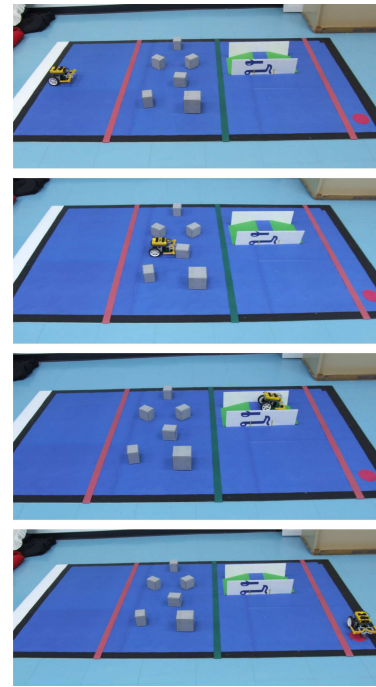


Figure 2.9: The LEGO robot used by the authors of [Wongwirat 08] moving in its environment (source: [Wongwirat 08]).

In Fig. 2.10, a UML activity diagram modelling the example of a robot exploring an area is depicted. The outputs are modelled as activities in the diagram, while the input conditions are represented by guard expressions of branches. As this type of diagram does not model states, the states of the example are not modelled directly, although one could say that the system is in a certain state as long as a certain activity is executed.

2.1.3 Petri Nets

Petri nets have been introduced by Carl Adam Petri in his dissertation, published in 1962 (see [Petri 62]). They are common for modelling dynamic, often distributed systems and by now, there are many different variants with different properties. One form are P/T nets, standing for “place/transition nets” (often called “Stellen/Transitions-Netze” in German).

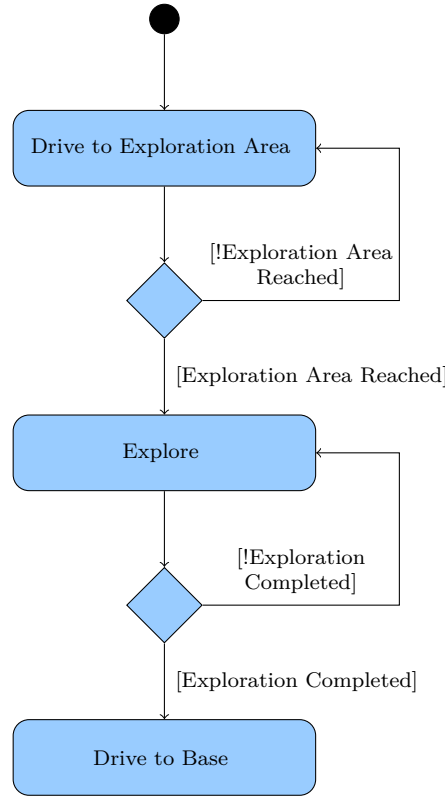


Figure 2.10: A UML activity diagram modelling the example of a robot exploring an area.

A P/T net is a graph consisting of two types of nodes: places (graphically represented by circles) and transitions (graphically represented by bars or rectangles). All edges in this graph connect one place with one transition, never two places or two transitions with each other. Places can be marked with tokens (graphically represented by dots). When a transition fires, a certain amount of tokens is taken from the preceding places and a certain amount of tokens is added to the succeeding places, depending on numbers attributed to the involved edges. To each place, the maximum amount of tokens that it can store at the same time (its capacity) is attributed. Any distribution of tokens is called a marking and represents the state of the system.

The author of [Reisig 85] first defines the term “finite net” (see Def. 2.7) and based on this the term “P/T net” (see Def. 2.8).

Definition 2.7: Finite Net

A *finite net* is a triple $N = (S, T; F)$ for which holds:

1. $S \cap T = \emptyset$, i.e. S and T are disjoint sets.
2. $F \subseteq (S \times T) \cup (T \times S)$ is a binary relation, the flow relation of N .

Definition 2.8: Place/Transition Net

A *place/transition net* (P/T net) is a 6-tuple $N = (S, T; F, K, M, W)$ for which the following holds:

1. $(S, T; F)$ is a finite net, and the elements of S and T are called places and transitions, respectively.
2. $K : S \rightarrow \mathbb{N} \cup \{\infty\}$ gives a (possibly unlimited) capacity for each place.
3. $W : F \rightarrow \mathbb{N} \setminus \{0\}$ attaches a weight to each arc of the net.
4. $M : S \rightarrow \mathbb{N} \cup \{\infty\}$ is the initial marking, respecting the capacities, i.e. $M(s) \leq K(s) \quad \forall s \in S$.

Other ways of defining P/T nets can also be found in the literature. For example, [Priese 08] uses two matrices to describe the weights attached to each arc. Besides the standard P/T nets, there are many other types of Petri nets for different applications. For example, in some Petri nets, a condition can be attached to a transition so that it can only fire if the condition holds (see the biscuit automaton presented in [Reisig 10] as an example).

In [Ziparo 06], it is described how Petri nets can be used to model complex action sequences. The approach is demonstrated using the example of a four-legged robot playing soccer. Figure 2.11 depicts a Petri net that models the task of searching for the ball and moving towards it until it has been reached using the three primitive actions *approachBall*, *trackBall*, and *seekBall*. The continuation of this work is published in [Ziparo 11].

According to Def. 2.8, the exploration example can be represented as follows:

$$\begin{aligned} S &= \{\text{Driving to Exploration Area}, \text{Exploring}, \text{Driving to Base}\} \\ T &= \{\text{Exploration Area Reached}, \text{Exploration Completed}\} \end{aligned}$$

$$\begin{aligned} F = \{ & (\text{Driving to Exploration Area}, \text{Exploration Area Reached}), \\ & (\text{Exploration Area Reached}, \text{Exploring}), (\text{Exploring}, \text{Exploration Completed}), \\ & (\text{Exploration Completed}, \text{Driving to Base}) \} \end{aligned}$$

$$K : \quad K(\text{Driving to Exploration Area}) = K(\text{Exploring}) = K(\text{Driving to Base}) = 1$$

$$\begin{aligned} W : \quad & W(\text{Driving to Exploration Area}, \text{Exploration Area Reached}) \\ &= W(\text{Exploration Area Reached}, \text{Exploring}) \\ &= W(\text{Exploring}, \text{Exploration Completed}) \\ &= W(\text{Exploration Completed}, \text{Driving to Base}) = 1 \end{aligned}$$

$$M : \quad M(\text{Driving to Exploration Area}) = 1; M(\text{Exploring}) = M(\text{Driving to Base}) = 0$$

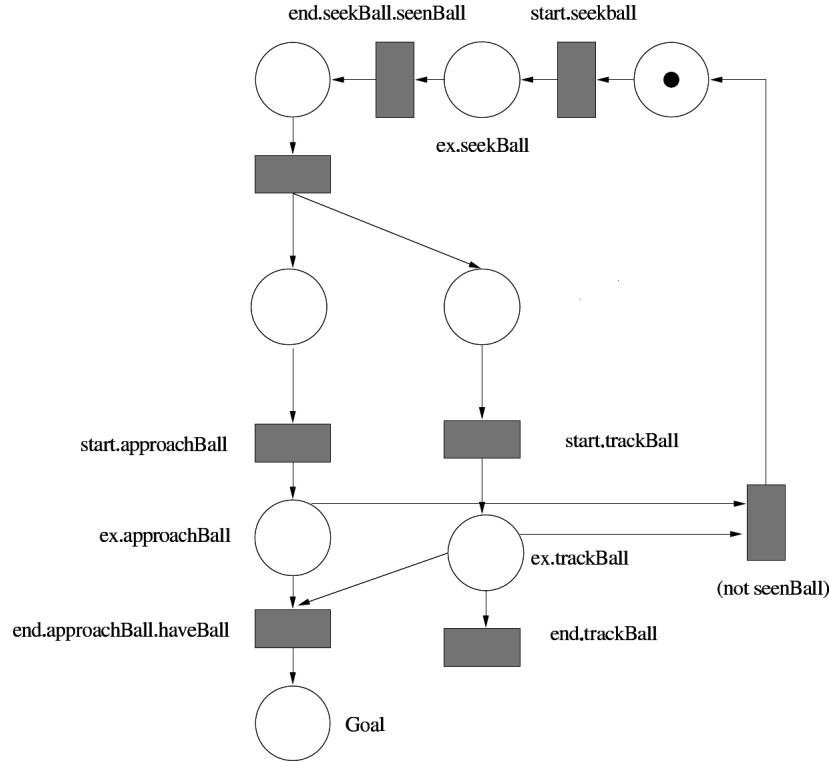


Figure 2.11: A Petri net modelling the task of seeking and approaching a ball, which shall be performed by an AIBO robot during a soccer play (source: [Ziparo 06]).

In this net, there is always exactly one token. When a place contains the token, this is interpreted as the system being in the state corresponding to the place in question. Figure 2.12 depicts the graphical representation of this Petri net.

Apart from the methods mentioned here, there are several other ways for specifying the sequential execution of tasks in a system. In the telecommunications sector, e.g., the Specification and Description Language (SDL) is used to describe systems by modelling them with connected augmented finite-state machines. It has been defined by the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T) in the recommendations Z.100 et seqq. (see [ITU 11] et seqq.). The SDL features a graphical as well as a textual format. Despite it originates from the telecommunications sector, it is today also used in other domains and would be suitable for defining task sequences in

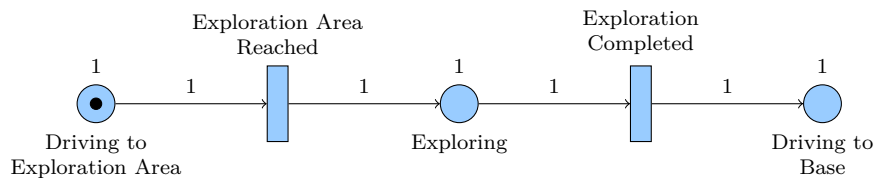


Figure 2.12: A P/T net modelling the example of a robot exploring an area. The figure depicts the initial marking: The place “Driving to Exploration Area” is marked, representing that the robot is driving to the exploration area.

robotics. In the area of industrial robotics, robots typically have to repeat sequences of actions again and again. These sequences are often defined using textual programming, but graphic input methods are also available.

All of the techniques described in this section can be used for defining task sequences. But while the textual definitions might be appealing to people with a background in mathematics, they are not the first choice of the typical developers of robot control systems. Furthermore, people from application domains such as the agricultural or construction sectors will not be able to use textual notations to describe what their robotic systems shall do. Graphical representations of sequences are therefore a much better choice. Especially the specification of sequences using graphical representations of finite-state machines is a method that can be used by technical personnel outside the area of robotics. This is an important aspect as often during the development of complex robots, people from robotics and people from an application domain have to team up. As FSMs can be textually specified, they can easily be used in automatic systems. Furthermore, FSMs are employed in numerous research studies in robotics. Hence, they are the method of choice for representing task sequences in the work at hand. Due to the reasons given in Sec. 2.1.1, Moore machines will be used.

2.2 Realisation of Sequences

While the previous section has dealt with methods for *representing* sequences, the purpose of this section is to present how sequences can be *realised* in the control systems of robots. The manner in which sequences are realised in a robot control system heavily depends on the type of the architecture according to which the system has been implemented. There are two opposing approaches to designing a robot control system: the deliberative and the reactive one. While the central technique of the former is using sensor data for creating complex plans that the robot shall execute, the latter focuses on a tight coupling of sensing and acting. Details about the two approaches along with examples can be found in Secs. A.1 and A.2, respectively. So-called hybrid robot control architectures (see Sec. A.3) combine a reactive layer with a deliberative one. A third layer is often used as an interface between the two. In this case, the architecture is referred to as three-layer architecture. Some examples of three-layer architectures are given in Sec. A.3. Their interfacing layer is able to execute complex tasks by hierarchically decomposing them into subtasks and employing components of the lower control level for executing them. This layer is therefore often referred to as sequencing layer or simply *sequencer*.

The RAP system described in Sec. A.3 can be used to realise such a sequencing layer. Its central component is the RAP interpreter, which is responsible for selecting the next task and splitting it into subtasks or executing a suitable primitive action. The RAP system is not only used in the architecture described in [Firby 89] and later work of the same author (see [Firby 94] and [Firby 95]), but also in systems developed by other researchers—for example ATLANTIS (see [Gat 92]) and 3T (see [Bonasso 97]). In [Gat 96], Erann Gat introduces the Execution Support Language (ESL), a set of extensions to Common Lisp, as an alternative to the RAP system in the sequencing layers of ATLANTIS and 3T. The main purpose of ESL is to be a useful programming tool and not a representation for reasoning or formal analysis.

In the original version of the hybrid architecture AuRA (see [Arkin 87]), a rule-based plan sequencer (called pilot) selected appropriate lower-level control components (called schemas) from a set of available components based on paths generated by a higher-level navigation component. The schemas controlled the robot in a reactive way and were responsible, for example, for moving the robot to a goal or stopping it depending on sensor input. The original sequencer was later replaced with a temporal sequencer that used finite-state machines for representing plans to execute (see [Arkin 94]).

In the architecture described in [Alami 98], the highest layer, called decision level, deals with task planning, generation of action sequences, and supervision. A special feature of this architecture is that the decision level contains two completely different components: the planner, which generates action sequences, and the supervisor, which monitors the progress of the execution and reacts to incoming events. The authors explain that the decision level can be split up into two or even more levels that operate on different representations. Each of these sublevels then contains a planner and a supervisor. In the example given in [Alami 98], the decision level contains two sublevels. The higher of the two generates partially ordered tasks with temporal constraints, while the lower transforms tasks into action sequences.

In case the sequencer is realised in a monolithic way, the logic which encodes the sequential execution of tasks is entirely stored in one single component. This component can easily become complex and difficult to maintain. In many cases, however, the sequencer is realised as a complete layer consisting of interconnected elements, i.e. its functionality is spread over several (potentially less complex) components. But in such architectures, there is usually a breach between the highest layer and the sequencer as well as between the sequencer and the lowest layer. This can complicate or limit the interaction of the three layers.

In order to avoid such a breach in advance, a type of architecture in which reactive and deliberative elements can be integrated in a homogeneous fashion has been chosen for realising the concepts of this thesis. This type of architecture is referred to as “behaviour-based architecture” and shall be defined in the following. There are many different definitions of what a behaviour-based architecture is. In the context of this thesis, the term refers to architectures which are built up of a number of interconnected components, the behaviours. Each of the behaviours is responsible for fulfilling a certain task.

[Brooks 91a] has identified a number of key aspects of robots with behaviour-based control systems (BBS):

Situatedness The robot directly deals with its real environment instead of with an abstract representation.

Embodiment The robot is a physical system that can influence the world. The effect of this influence is in turn perceived by its sensor systems.

Intelligence The intelligence of the robot does not only originate from its control system, but also from the data processing in its sensors and the physical interaction with its environment.

Emergence The intelligence of the robot is partly created by interactions between the robot and its environment or by (possibly indirect) interactions of the components of its control system.

Situatedness and embodiment seem to describe similar properties. However, they are clearly distinct. [Brooks 91b] gives two illustrative examples of the differences between the two.

On the one hand, the four above-mentioned aspects often lead to behaviour-based robots appearing much more vivid than the ones controlled by purely deliberative systems, which need a considerable amount of time for constructing a complex world model. On the other hand, the emergence of a robot behaviour from the interaction of the robot with its environment and from the indirect interaction of single components makes the prediction of the overall system behaviour difficult and complicates verification significantly. Therefore, a part of this thesis is dedicated to describing a verification concept that copes with the high degree of distribution in behaviour-based systems (see Chap. 4).

It is common for behaviour-based systems to contain reactive elements. But contrary to many purely reactive architectures, the calculations in behaviour-based architectures are not limited to simple mappings of input to output values. Furthermore, the author of [Matarić 97] argues that behaviour-based architectures are able to store and operate on complex data structures. However, in [Kortenkamp 08], it is explained that it is controversial in the field how much state information should be stored in the lowest layer of a control system. An extreme position was formulated by Rodney Brooks in [Brooks 90], where he stated “that the world is its own best model” and that therefore, a control system should be based on the real world and not on abstract representations of it. The author of [Gat 93] suggests ensuring that the internal state of a robot is at least usually correct by storing the state at a high level of abstraction and by grounding the immediate control of the robot on current sensor data.

The definition of behaviour-based architectures used in this thesis (see below), however, allows for behaviours storing any type of world representation, no matter how complex it is. But in contrast to many purely deliberative architectures, which maintain a central representation of the robot’s environment that all components (have to) operate on, behaviours can deal with distributed data structures that usually represent only parts or certain aspects of the environment.

Definition 2.9: Behaviour-Based Robot Control Architecture

A *behaviour-based robot control architecture* is an architecture consisting of a number of distributed, interacting components (the behaviours). Each behaviour can have an arbitrary complexity in terms of processing and storage of sensor data and tries to influence the robot’s actions in a certain way. The overall robot behaviour results from the interaction and the combined outputs of all behaviours.

The distribution of a control system into a number of independent components yields several advantages of the behaviour-based architectures over monolithic approaches: The single components can be developed, implemented, and tested independently, allowing for distributed work on the control system and partial integration for test purposes. The distribution of the overall functionality typically results in components with lower complexity, which facilitates verification and validation. Furthermore, it fosters component reuse—which can get complicated to impossible in monolithic systems. The distribution into single components can also increase the robustness of a system in case some components are redundant: If one behaviour fails, another one might still be fully operational, enabling

the system to fulfil its task despite the failure of one component. As behaviour-based systems often contain reactive components, they are able to react fast to changes in the sensor data and are thus well-suited for highly dynamic environments.

Meanwhile, there are numerous concepts and approaches for behaviour-based control—the motor schema-based approach (see [Arkin 89]), DAMN (see [Rosenblatt 97]), and the use of fuzzy logic (see [Saffiotti 97]), only to name a few. Robots controlled by behaviour-based systems typically feature many biologically inspired aspects with respect to their software as well as their hardware. For example, some of them resemble insects (see [Brooks 89a] and [Brooks 89b]). The Springer Handbook of Robotics (see [Siciliano 08]) dedicates a whole chapter to behaviour-based systems (see [Matarić 08]) and a part of the chapter about robotic systems architectures and programming also deals with them (see [Kortenkamp 08]). More recent work is presented in [Langosz 13], for example, which describes the representation of behaviours as directed graphs.

For the realisation of complex tasks with a behaviour-based system, it is necessary that the operation of behaviours can be sequenced somehow. These sequences need not be linear. The authors of [Gat 94], for example, elaborate on the application of so-called conditional sequencing for controlling autonomous mobile robots. They use the term to describe sequences that are not necessarily linear, but differ depending on the situation. By evaluating all possible combinations of conditions, a conditional sequence can be linearised, resulting in a number of linear sequences.

In the following, it is assumed that one behaviour is only capable of realising exactly one task. It is further assumed that for each task, there is only one behaviour that can fulfil it. The first assumption is in line with the common notion that each behaviour is responsible for fulfilling a certain task. The second one, however, contradicts the concept of having several (redundant) behaviours that take care of a single task. However, on a higher level of abstraction, redundant behaviours can be combined to one larger behaviour (see comments about grouping behaviours below) so that the contradiction is resolved. In order to be able to execute a task, a behaviour must somehow get active. The sequential execution of tasks then corresponds to the associated behaviours getting active in a certain sequence. Such a sequence shall be called “behaviour activity sequence” and is formally defined according to Def. 2.10 as follows:

Definition 2.10: Behaviour Activity Sequence

Be \mathbb{B} a set of behaviours. Then a (finite) *behaviour activity sequence* is a function $s : \{0, \dots, n-1\} \rightarrow \mathbb{B} \times \mathbb{R}$ with elements $s(i) = s_i$ defined as $s_0 = (B_{j_0}, t_0), \dots, s_{n-1} = (B_{j_{n-1}}, t_{n-1})$ for which the following holds:

1. i is an index for the elements of s , i.e. $0 \leq i \leq n-1$.
2. j_i are indices into the set of behaviours \mathbb{B} , i.e. $0 \leq j_i \leq |\mathbb{B}| - 1$.
3. $t_i \in [0, T] \subset \mathbb{R}$ denote distinct points in a time interval ranging from 0 to T with $i < k \implies t_i < t_k$.
4. The activity of behaviour B_{j_i} is denoted with a_{j_i} .
5. B_{j_i} gets active at t_i , i.e. $\forall s_i = (B_{j_i}, t_i) :$
 $(\exists \delta > 0 : \forall \varepsilon \text{ with } 0 < \varepsilon \leq \delta : a_{j_i}(t_i - \varepsilon) = 0) \wedge (a_{j_i}(t_i) = 1)$.

The exploration example that has already been mentioned several times shall be picked up again to illustrate Def. 2.10. The control system of the robot shall have three behaviours available: *Drive to Exploration Area*, *Explore*, and *Drive to Base*. In this example, $n = 3$ and $\mathbb{B} = \{B_0, B_1, B_2\}$ with $B_0 = \text{Drive to Exploration Area}$, $B_1 = \text{Explore}$, and $B_2 = \text{Drive to Base}$. As a result, the sequence s representing the given behaviour activities in the correct order is defined as follows:

$$\begin{aligned} s_0 &= (B_0, t_0) = (\text{Drive to Exploration Area}, t_0) \\ s_1 &= (B_1, t_1) = (\text{Explore}, t_1) \\ s_2 &= (B_2, t_2) = (\text{Drive to Base}, t_2) \end{aligned}$$

Again, the exact times t_i are insignificant in many cases. However, it is important that $i < k \implies t_i < t_k$, i.e. that the points at which the behaviours get active are ordered in a sequence. Therefore, the behaviour activity sequence of the example can be written in a shortened form:

$$\begin{aligned} s_0 &= B_0 = \text{Drive to Exploration Area} \\ s_1 &= B_1 = \text{Explore} \\ s_2 &= B_2 = \text{Drive to Base} \end{aligned}$$

As has been illustrated using the example, behaviour activity sequences can be defined formally based on Def. 2.10. However, for the practical application during the development of a robot control system, it is often more useful to define *task* sequences (cp. Sec. 2.1), which are then realised as behaviour activity sequences in the behaviour-based control system of a robot.

There are different approaches to realising sequences in behaviour-based systems. Two major concepts (centralised and decentralised) can be distinguished. They are presented in the following two sections along with examples of architectures in which they are implemented.

2.2.1 Centralised Realisation of Sequences in Behaviour-Based Systems

The centralised realisation of sequences in behaviour-based systems resembles the idea of the sequencer in three-layer architectures: Special components trigger the sequential execution of tasks by activating or deactivating other components. In behaviour-based systems, these special components are coordinating behaviours that take care of activating other behaviours depending on sensor inputs. Each of these coordinating behaviours contains the logic for triggering a sequence.

A special case of such coordinating behaviours are *central pattern generators* (CPGs), which are components that can be used to issue synchronised periodic motions of actuators. They are often realised as self-oscillating systems that do not feature any inputs except for parameters that can be used to tune their output. CPGs are typically used in biologically inspired robots, e.g. multilegged systems. In the following, an example of the use of CPGs is presented. Further comments and references to literature regarding CPGs can be found in [Kajita 08].

The author of [Luksch 10] describes the behaviour-based control of a dynamically walking bipedal robot. He introduces so-called *spinal pattern generators* (SPGs). These are CPGs that correspond to pattern generators located in the spinal cord. The proposed control approach has been realised using the behaviour-based architecture iB2C¹. This architecture has been developed—and still is enhanced—at the Robotics Research Lab² of the Department of Computer Science³ at the University of Kaiserslautern⁴, Germany. It has been implemented in the robotics framework MCA2-KL (see Sec. B.1) and its downward compatible successor FINROC (see Sec. B.2). The iB2C is described extensively in [Proetzsch 10]. As it is a basis for the work at hand, a brief description is given in the following.

The central component of the iB2C is a behaviour. Its symbol is depicted in Fig. 2.13. All iB2C behaviours share a common interface, which consists of *stimulation* $s \in [0, 1]$ (for gradually enabling a behaviour), *inhibition* $i \in [0, 1]$ (for gradually disabling it), *activity* $a \in [0, 1]$ (corresponding to the degree of influence a behaviour intends to have in a network), and *target rating* $r \in [0, 1]$ (indicating the behaviour's dissatisfaction with the current situation). Stimulation s and inhibition i are combined to the *activation* $\iota = s \cdot (1 - i)$. The inhibitory input can receive an *inhibition vector* $\vec{i} = (i_0, \dots, i_{k-1})^T$ so that multiple inhibitory links can be connected. The behaviour's inhibition is then calculated as $i = \|\vec{i}\|_\infty$. Furthermore, a behaviour can possess q so-called *derived activities* $\vec{a} = (\underline{a}_0, \dots, \underline{a}_{q-1})^T$ with $\underline{a}_i \leq a \ \forall i \in \{0, \dots, q-1\}$, which allow for transferring only a part of its activity to the network. Together with a , they build the *activity vector* $\vec{a} = (a, \underline{a})^T$. Stimulation, inhibition, activation, activity, and target rating are called *behaviour signals*. Their value range is limited to $[0, 1]$. Besides this common interface, each behaviour can have a specialised interface consisting of the *input vector* $\vec{e} \in \mathbb{R}^m$ and the *output vector* $\vec{u} \in \mathbb{R}^n$. The elements of the input and output vectors are called *control values*. In contrast to the behaviour signals, their value ranges are not limited, i.e.

¹iB2C: integrated Behavior-Based Control

²website: <http://rrlab.cs.uni-kl.de/>

³website: <http://cs.uni-kl.de/>

⁴website: <http://uni-kl.de/>

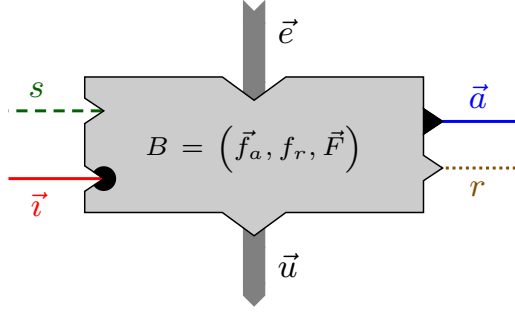


Figure 2.13: The general symbol of an iB2C behaviour (s : stimulation, $\vec{\iota}$: inhibition vector, \vec{a} : activity vector, r : target rating, \vec{e} : input vector, \vec{u} : output vector, \vec{f}_a : function calculating \vec{a} , f_r : function calculating r , \vec{F} : function calculating \vec{u}).

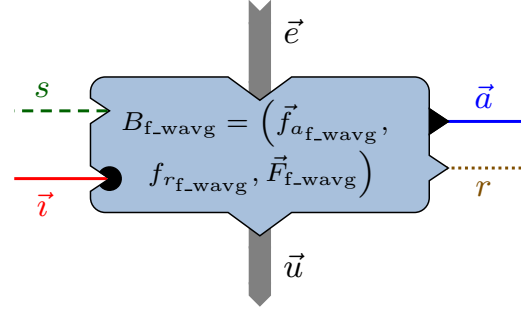


Figure 2.14: The symbol of a fusion behaviour realising a weighted average fusion. The symbol indicates that a fusion behaviour—as any other behaviour—has an interface consisting of behaviour signals and control values.

they can transfer any kind of data that can be represented as a sequence of real numbers. Formally, a behaviour is defined as $B = (\vec{f}_a, f_r, \vec{F})$, with $\vec{f}_a : \mathbb{R}^m \times [0, 1] \rightarrow [0, 1] \times [0, 1]^q$ and $\vec{f}_a(\vec{e}, \iota) = \vec{a}$ being the *activity function*, f_r with $f_r : \mathbb{R}^m \rightarrow [0, 1]$ and $f_r(\vec{e}) = r$ being the *target rating function*, and \vec{F} with $\vec{F} : \mathbb{R}^m \times [0, 1] \rightarrow \mathbb{R}^n$ and $\vec{F}(\vec{e}, \iota) = \vec{u}$ being the *transfer function*. There is no limitation of the complexity of the transfer function, i.e. it can be anything from a simple linear function to a complex image processing algorithm.

As can be seen in Fig. 2.13, the signal lines have different colours and styles. Stimulating connections are depicted by dashed green lines, while inhibition links are represented by solid red lines. General transfers of activity values are visualised using solid blue lines, while transmissions of target rating values are symbolised by dotted brown lines. Links for exchanging control values are depicted by solid dark grey lines. These styles and colours are used throughout this thesis in figures depicting iB2C networks.

Apart from the value ranges, there are a number of further restrictions of the behaviour signals, which are defined in [Proetzsch 10] as *principles*. The principle relevant for the work at hand states that the activity a_B of a behaviour B is limited by the behaviour's activation ι_B : $a_B \leq \iota_B$. With $\underline{a}_i \leq a \ \forall i \in \{0, \dots, q-1\}$ (see above), the principle indirectly also states that $\underline{a}_i \leq \iota \ \forall i \in \{0, \dots, q-1\}$, i.e. the derived activities are also limited by a behaviour's activation. It is important not to confuse the two (clearly distinct) terms *activation* and *activity*. Furthermore, if a behaviour's activation is > 0 , the behaviour is said to be *activated* to a certain degree, whereas a behaviour with activity > 0 is said to be *active* to a certain degree.

There are various methods for connecting iB2C behaviours in a network. For example, the activity output of a behaviour B_0 can be connected to the stimulation input of a behaviour B_1 . B_0 then stimulates B_1 with its activity (see Fig. 2.15). By connecting the activity output of B_0 to the inhibition input of B_1 , an inhibitory link can be established so that B_0 inhibits B_1 with its activity (see Fig. 2.16). The iB2C features a special coordination behaviour: the *fusion behaviour*. A fusion behaviour B_{Fusion} can be used to combine the outputs of n_c competing behaviours B_{Input_d} ($d = 0, \dots, n_c - 1$) with activities a_{Input_d} , target ratings r_{Input_d} , and output vectors \vec{u}_{Input_d} according to one of three different fusion

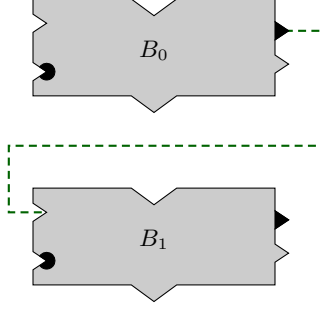


Figure 2.15: Behaviour B_0 stimulates behaviour B_1 with its activity.

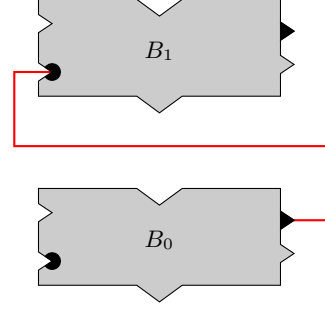


Figure 2.16: Behaviour B_0 inhibits behaviour B_1 with its activity.

methods. The input vector \vec{e}_{Fusion} of B_{Fusion} is composed of a_{Input_d} , r_{Input_d} , and \vec{u}_{Input_d} with $d = 0, \dots, n_c - 1$. The type of the fusion method (maximum, weighted average, and weighted sum) determines $\vec{f}_{a_{\text{Fusion}}}$, $f_{r_{\text{Fusion}}}$, and \vec{F}_{Fusion} of the fusion behaviour:

Maximum Fusion

$$B_{\text{Fusion}} = B_{\text{f_max}} = (\vec{f}_{a_{\text{f_max}}}, f_{r_{\text{f_max}}}, \vec{F}_{\text{f_max}}) \quad (2.1)$$

$$\vec{f}_{a_{\text{f_max}}}(\vec{e}_{\text{f_max}}, \iota_{\text{f_max}}) = \max_d (a_{\text{Input}_d}) \cdot \iota_{\text{f_max}} \quad (2.2)$$

$$f_{r_{\text{f_max}}}(\vec{e}_{\text{f_max}}) = r_{\text{Input}_g} \text{ with } g = \underset{d}{\operatorname{argmax}} (a_{\text{Input}_d}) \quad (2.3)$$

$$\vec{F}_{\text{f_max}}(\vec{e}_{\text{f_max}}) = \vec{u}_{\text{Input}_g} \text{ with } g = \underset{d}{\operatorname{argmax}} (a_{\text{Input}_d}) \quad (2.4)$$

Weighted Average Fusion

$$B_{\text{Fusion}} = B_{\text{f_wavg}} = (\vec{f}_{a_{\text{f_wavg}}}, f_{r_{\text{f_wavg}}}, \vec{F}_{\text{f_wavg}}) \quad (2.5)$$

$$\vec{f}_{a_{\text{f_wavg}}}(\vec{e}_{\text{f_wavg}}, \iota_{\text{f_wavg}}) = \frac{\sum_{j=0}^{n_c-1} a_{\text{Input}_j}^2}{\sum_{k=0}^{n_c-1} a_{\text{Input}_k}} \cdot \iota_{\text{f_wavg}} \quad (2.6)$$

$$f_{r_{\text{f_wavg}}}(\vec{e}_{\text{f_wavg}}) = \frac{\sum_{j=0}^{n_c-1} (a_{\text{Input}_j} \cdot r_{\text{Input}_j})}{\sum_{k=0}^{n_c-1} a_{\text{Input}_k}} \quad (2.7)$$

$$\vec{F}_{\text{f_wavg}}(\vec{e}_{\text{f_wavg}}) = \frac{\sum_{j=0}^{n_c-1} (a_{\text{Input}_j} \cdot \vec{u}_{\text{Input}_j})}{\sum_{k=0}^{n_c-1} a_{\text{Input}_k}} \quad (2.8)$$

Weighted Sum Fusion

$$B_{\text{Fusion}} = B_{\text{f_wsum}} = (\vec{f}_{a_{\text{f_wsum}}}, f_{r_{\text{f_wsum}}}, \vec{F}_{\text{f_wsum}}) \quad (2.9)$$

$$\vec{f}_{a_{\text{f_wsum}}}(\vec{e}_{\text{f_wsum}}, \iota_{\text{f_wsum}}) = \min \left(1, \sum_{j=0}^{n_c-1} \frac{a_{\text{Input}_j}^2}{\max_d (a_{\text{Input}_d})} \right) \cdot \iota_{\text{f_wsum}} \quad (2.10)$$

$$f_{r_{\text{f_wsum}}}(\vec{e}_{\text{f_wsum}}) = \frac{\sum_{j=0}^{n_c-1} (a_{\text{Input}_j} \cdot r_{\text{Input}_j})}{\sum_{k=0}^{n_c-1} a_{\text{Input}_k}} \quad (2.11)$$

$$\vec{F}_{\text{f_wsum}}(\vec{e}_{\text{f_wsum}}) = \sum_{j=0}^{n_c-1} \frac{a_{\text{Input}_j} \cdot \vec{u}_{\text{Input}_j}}{\max_d (a_{\text{Input}_d})} \quad (2.12)$$

Figure 2.14 depicts the symbol of a weighted average fusion behaviour. The symbol of a fusion behaviour realising a maximum fusion has a darker blue, while the symbol of a fusion behaviour realising a weighted sum fusion has a lighter blue. In order to distinguish fusion behaviours from other types of behaviours, the names of the former are prefixed with “(F)” in textual descriptions. In figures depicting iB2C networks, the “(F)” is often left out as the different colours are sufficient for distinguishing different types of behaviours.

Having a fusion behaviour is not a unique feature of the iB2C. For example, the authors of [Nicolescu 07] describe a so-called *fusion primitive* that combines the outputs of several behaviours. But in contrast to the iB2C fusion behaviour, the fusion primitive only features one fusion method. It contains several sets of fixed weights that are used to create a linear combination of the outputs of the behaviours connected to the fusion primitive. Which set is chosen depends on a vector that is built from the active/not active outputs of all connected behaviours. Providing three different fusion methods like in the iB2C offers more flexibility to the developer.

As explained in [Pirjanian 99], there are various methods for realising the coordination between behaviours. [Proetzsch 10] describes how to realise many of them using the iB2C. Figure 2.17 shows how a priority-based arbitration of two behaviours B_0 and B_1 can be realised using an inhibitory link and a maximum fusion behaviour. In the depicted network, B_0 can raise its activity to inhibit B_1 so that a_1 falls below a_0 and—as a result— B_0 will overrule B_1 during the maximum fusion.

The iB2C features a mechanism for abstracting several behaviours into a single component, the *behaviour group* (also referred to as *behavioural group*). With this mechanism, hierarchical behaviour networks can be created. A behaviour group features the same interface as all behaviours, i.e. it can be used in a behaviour network in the same way as a normal behaviour. It usually contains a behaviour coordinating the other behaviours contained in the group. The group forwards its stimulation and inhibition ports to this coordinating behaviour, which in turn forwards its activity and target rating to the corresponding ports of the group. Figure 2.18 provides an example. As can be seen, the symbol of a group is similar to the one of a standard behaviour, but features a double line as boundary. The names of groups are prefixed with “(G)” in textual descriptions. In figures, this prefix

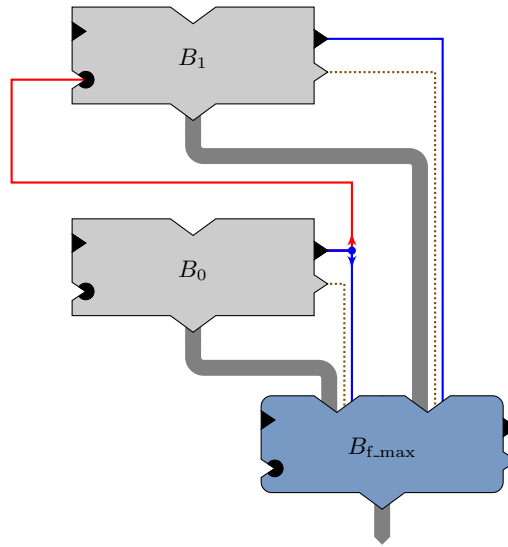


Figure 2.17: A simple behaviour network realising a priority-based arbitration. The inhibitory link from B_0 to B_1 guarantees that B_0 can overrule B_1 .

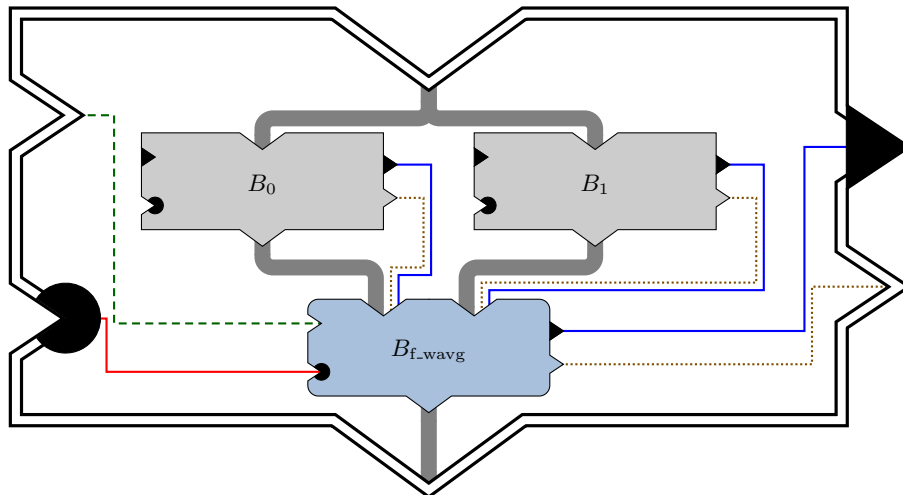


Figure 2.18: A behaviour group combining two behaviours B_0 and B_1 . The coordinating behaviour here is a weighted average fusion behaviour. It receives the stimulation and inhibition of the group and provides the group's activity and target rating.

is often left out as the different styles of their boundary lines allow for distinguishing standard behaviours from behaviour groups.

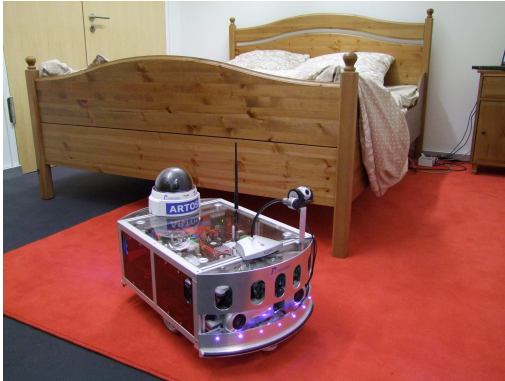


Figure 2.19: The mobile indoor robot ARTOS in a mock-up of an assisted living environment.



Figure 2.20: The autonomous off-road vehicle RAVON in the Palatinate Forest.

The iB2C has been applied in the control systems of a number of robots, ranging from small indoor robots to large outdoor vehicles. In Figs. 2.19 and 2.20, two examples of robots controlled by iB2C behaviour networks are shown. ARTOS⁵ is a small vehicle developed for research in the area of assisted living. It is described in [Armbrust 07] as well as in [Koch 08] (mechatronics system and collision avoidance), [Mehdi 09] (mapping and navigation), and [Armbrust 11b] (all aspects). The autonomous off-road vehicle RAVON⁶ (see [Armbrust 09a] and [Armbrust 10a] for an overview) features a sophisticated control system that contains over 500 iB2C behaviours. Parts of this system are used to illustrate the design concept (see Sec. 3.1.2) as well as the verification concept (see Sec. 4.3.1) described in this thesis.

As already mentioned above, the behaviour-based control of a dynamically walking bipedal robot that is described in [Luksch 10] has been implemented using the iB2C. For example, the SPG realising the cyclic walking has been realised as an iB2C behaviour. In Figure 2.21, the state machine defining the different phases of stable standing is depicted. The author comments that a drawback of his approach “is the absence of a strong timing to create periodic movement” and explains that “the periodicity emerges from the robots[sic] interaction with the environment”. The task of the SPGs is to switch between different motion phases by stimulating special control units, which in turn stimulate components that realise reflexes. Figure 2.22 shows the structure of the subsystem responsible for stable standing with the components involved in the first phase (ground adaptation) being highlighted. In Fig. 2.23, three images depict experiments with the simulated biped. While walking, the biped is exposed to different types of disturbances (downhill slope, step, and external force).

In summary, the iB2C offers a number of mechanisms that facilitate building complex behaviour networks: a standard behaviour interface, different types of behaviour interaction, and a mechanism for creating hierarchical networks. However, while it is possible to create sequences of behaviour activation, there is mainly support for the centralised realisation,

⁵ARTOS: Autonomous Robot for Transport and Service

⁶RAVON: Robust Autonomous Vehicle for Off-road Navigation

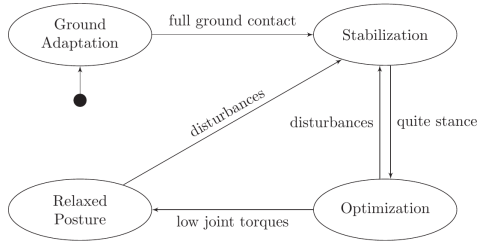


Figure 2.21: The finite-state machine representing the SPG that is responsible for the different phases of stable standing (source: [Luksch 10]).

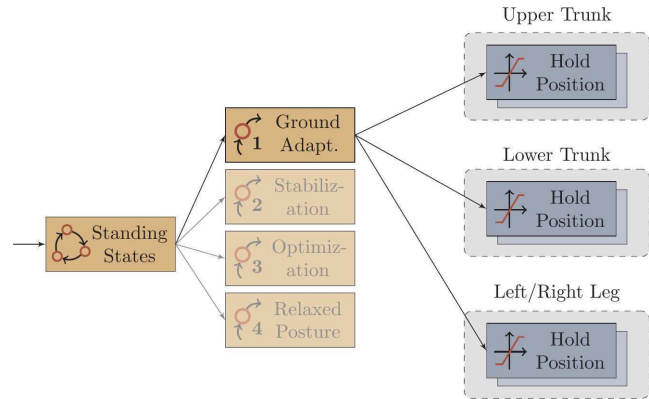


Figure 2.22: The structure of the subsystem that is responsible for stable standing. The components needed for the first phase—ground adaptation—are highlighted (source: [Luksch 10]).

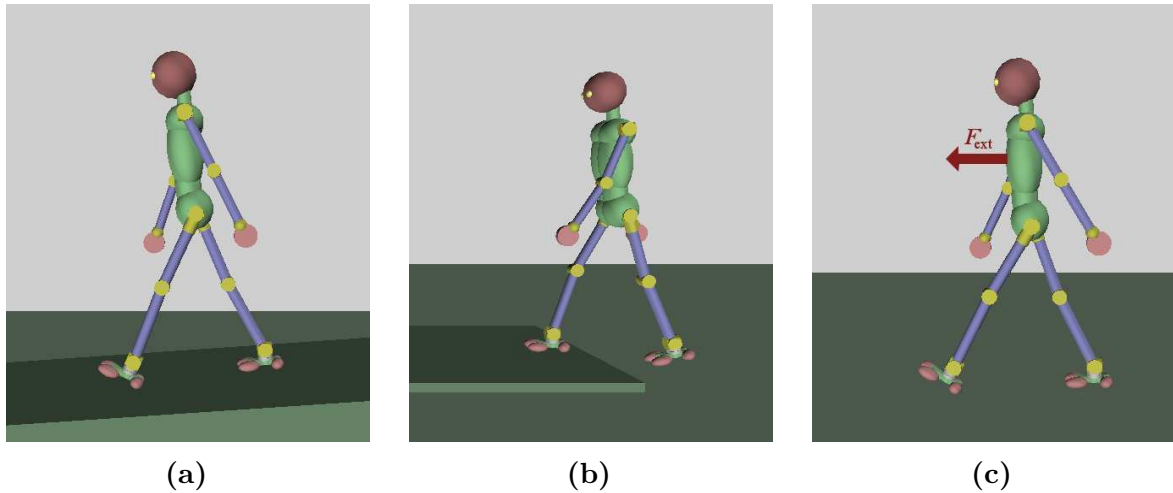


Figure 2.23: The simulated biped is exposed to different types of disturbances while walking: a downhill slope (Fig. a), a step (Fig. b), and an external force acting on its torso (Fig. c) (source: [Luksch 10]).

which is based on behaviours that implement CPGs. But the use of CPGs for realising behaviour activity sequences has a major disadvantage: CPGs concentrate the complete logic of the sequential execution of subtasks in one component, which thus can easily get very complex and hence difficult to adapt or enhance. The decentralised realisation of sequences (see Sec. 2.2.2) does not have this disadvantage, but there is no dedicated support for it in the iB2C. The only option to realise sequences in a decentralised way that is directly offered by the iB2C is to use stimulating and inhibiting links between behaviours (cp. the method described in Sec. 2.2.2.1).

Furthermore, the iB2C only features little support for the verification of behaviour networks. The authors of [Proetzsch 07] describe how a group of behaviours can be formally verified by implementing it in the synchronous language Quartz (see [Schneider 09]) and using the model checking capabilities of the Averest verification framework (see [Schneider 05]). In an application example, C code was automatically generated from the Quartz code after the verification step and integrated into the control system of RAVON. As a result, it could be guaranteed that the behaviours in question operated as desired. A major drawback of this approach is that the behaviours to be verified have to be (re-)implemented in Quartz. The approach did not allow for directly checking the correct operation of an already existing behaviour network.

Chapters 3 and 4 approach these limitations of the iB2C and describe concepts for designing complex iB2C networks realising sequences of tasks and for the verification of iB2C networks using model checking.

2.2.2 Decentralised Realisation of Sequences in Behaviour-Based Systems

In the previous section, the centralised realisation of sequences in behaviour networks has been presented. Their concentration of the logic for executing sequences in single behaviours in a way contradicts the concept of distributing functionality over a number of behaviours in a network. In the following, it will be explained how behaviour activity sequences can be realised in behaviour-based systems without concentrating the sequencing logic in special behaviours. Instead, the logic is distributed over several behaviours, in other words directly encoded in the behaviour network. There are two ways of realising this: without (see Sec. 2.2.2.1) or with (see Sec. 2.2.2.2) special inter-behaviour connections.

2.2.2.1 Decentralised Realisation of Sequences Without Using Special Inter-Behaviour Connections

When encoding behaviour activity sequences in a behaviour network in a decentralised way without using special inter-behaviour connections, there are no connecting elements whose task is to realise the sequential activation of behaviours. There are neither coordinating elements nor special links between behaviours that establish a temporal succession of behaviour activations. In such networks, sequential behaviour activities only result from the interaction of a robot with its environment and prioritisation among its behaviours. Thus, this way of realising sequences can be achieved even with simple behaviour architectures that only feature basic activation and inhibition links.

An example of the application of this sequencing technique is given in [Winfield 09]. Its author describes how a simple foraging task defined by a finite-state machine can be realised

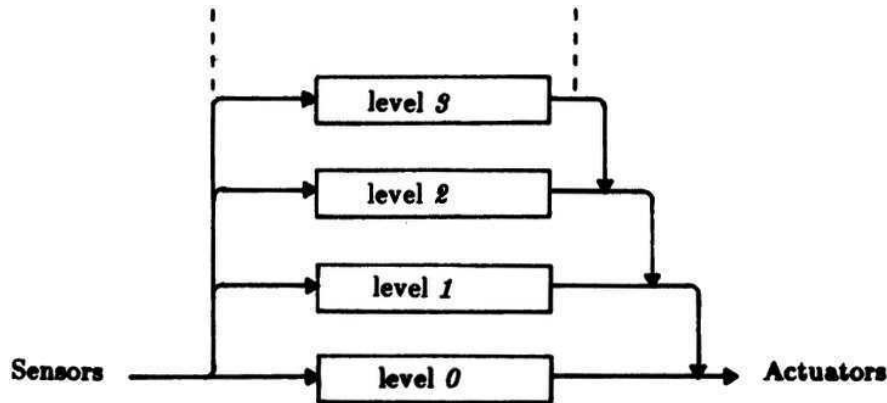


Figure 2.24: Four different layers of control. Higher layers can subsume the functionality of lower layers in case they want to control the robot. All layers have direct access to the sensors (source: [Brooks 86]).

as a control system with five levels that is implemented using the so-called *subsumption architecture*. This architecture was invented by Rodney Brooks (see [Brooks 86]) and is said to be the very first behaviour-based architecture. It is the result of decomposing the problem a robot shall solve vertically into *levels of competence*, each constituting a certain capability of the robot (collision avoidance, wandering, exploration, ...). In the control system, a layer is created corresponding to each level of competence. The author of [Brooks 86] describes this as an iterative process, starting with the lowest layer and going up to the highest one. Each layer is able to examine data of the lower layer and—by injecting data into the lower layer—to suppress the normal data flow. Figure 2.24 illustrates this concept, which is an example of priority-based arbitration according to the taxonomy given in [Pirjanian 99].

Each layer in a subsumption-based robot control system is built up of so-called *modules*, asynchronously running processes realised by finite-state machines augmented with data structures. The interface of a module consists of input and output ports used to connect modules. Input signals can be suppressed (i.e. overwritten by an input from a different module) and output signals can be inhibited (i.e. blocked). Furthermore, each module features a reset input with which the contained finite-state machine can be set to its initial state. Figure 2.25 depicts a module, while Fig. 2.26 depicts the lowest layer of a control system which makes the robot keep away from obstacles.

While the subsumption architecture represents a milestone in the development of robot control architectures and allows for building robots that show a very intelligent behaviour despite their simplicity, it has certain weaknesses: Except from the layers, it does not offer any abstraction technique like combining a set of behaviours into a group. As the layers directly determine the priority that behaviours have in the network, the assignment of behaviours to layers can only partially be used to realise different levels of abstraction. Other architectures, like the iB2C (see Sec. 2.2.1) and the USC behaviour architecture (see Sec. 2.2.2.2), by contrast, offer behaviour grouping. Furthermore, there are no fixed guidelines for a developer about how to realise the layers. In [Brooks 86], it is explained that each single layer is (at least to some extent) structured in the “traditional manner” (i.e. separated into functional modules). However, there is no explanation on how to implement

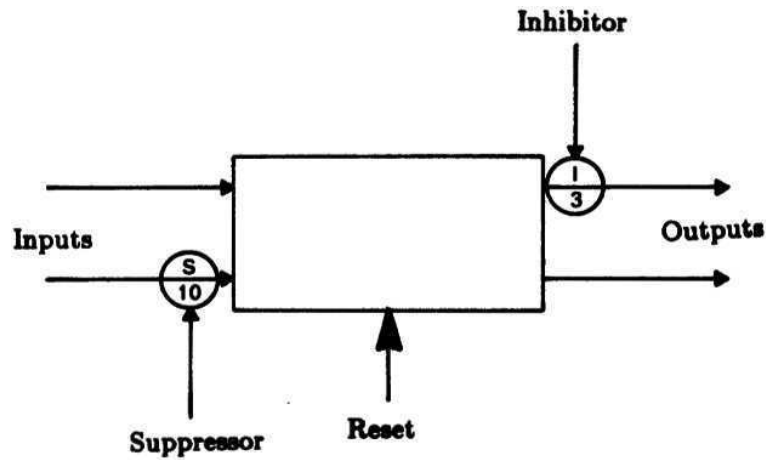


Figure 2.25: The symbol of a module in the subsumption architecture. The circles are connection points for suppressing (S) and inhibiting (I) signals. The numbers indicate the duration of the suppression and the inhibition, respectively. The reset input can be used to set the contained augmented finite-state machine to its initial state (source: [Brooks 86]).

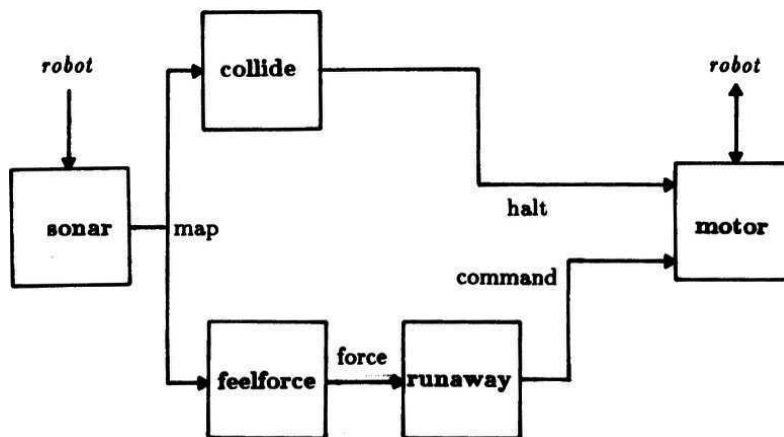


Figure 2.26: The lowest layer of a control system for a mobile robot. It realises collision-free motion (source: [Brooks 86]).

a system that realises complex sequences of actions. In Chap. 3, it is demonstrated how this can be done using the iB2C. A central aspect of behaviour-based systems is the emergence of an overall system behaviour from the interaction of single behaviours. This aspect can make it hard to predict what the overall system behaviour will be. In [Brooks 91a], Brooks mentions this problem for the case of interacting robots, but it also exists within single systems. Verification that targets the interaction of behaviours is a technique to solve this problem. Chapter 4 of the work at hand deals with this.

Figure 2.27 depicts the finite-state machine that defines the foraging task presented in [Winfield 09]. The realisation as a subsumption network is shown in Fig. 2.28. In this example, there are no coordinating behaviours or connections dedicated explicitly to realising a sequence. Instead, the sequential execution emerges from the prioritisation of the involved behaviours as well as the robot's actions in its environment. As the example shows, it is possible to create behaviour networks that realise the sequencing of behaviour activities with an architecture offering only a comparably simple mechanism for interconnecting behaviours. However, this method is not suitable for more than rather simple sequences as it does not feature support for creating more complex connections between behaviours. Therefore, it does not offer any assistance for developers that have to realise complex tasks consisting of sequences of subtasks.

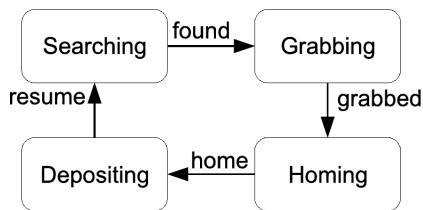


Figure 2.27: A finite-state machine with four states illustrating a simple foraging task (source: [Winfield 09]).

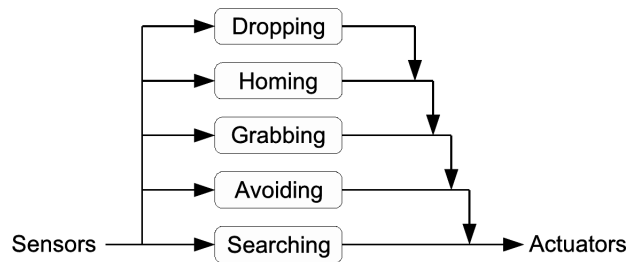


Figure 2.28: A subsumption network realising a simple foraging task, extended with collision avoidance functionality (source: [Winfield 09]).

2.2.2.2 Decentralised Realisation of Sequences Using Special Inter-Behaviour Connections

A more sophisticated way of realising behaviour activity sequences in a behaviour network in a decentralised way is the use of special inter-behaviour connections. In contrast to the approach described in the previous section, the sequential activation of behaviours is not merely a product of behaviour prioritisation and the robot's actions in its environment. Instead, there are dedicated connections between behaviours that can enforce a specific temporal succession between the activation of behaviours. There may even be coordinating behaviours that help establishing such connections.

In [Maes 90], an architecture is presented that uses this technique. The architecture realises behaviour activity sequences using the spreading of activation among behaviours (based on mutual activating and inhibiting of behaviours). This mechanism alone would characterise the architecture as a representative of the technique described in the previous section. However, the activation and inhibition of behaviours is done using so-called

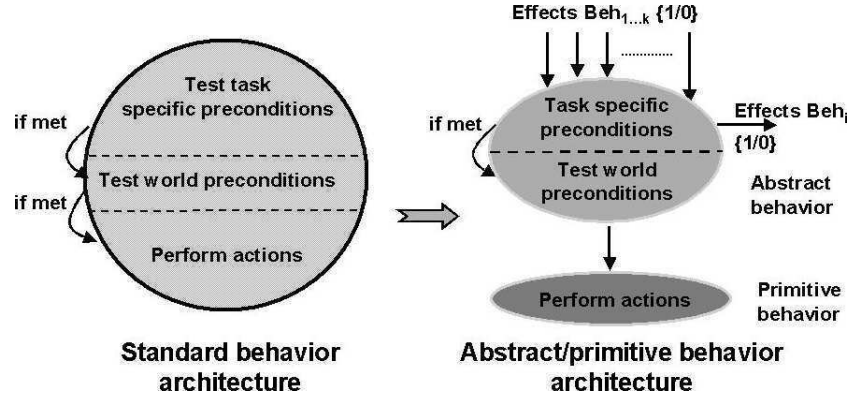


Figure 2.29: The difference between the usual monolithic structure of a behaviour and the structure of a behaviour in the USC behaviour architecture, which is split into an abstract part for checking preconditions and a primitive part for the actual task execution (source: [Nicolescu 02]).

successor, predecessor, and conflictor links, which indicate whether one behaviour precedes, succeeds, or conflicts with another. The paper describes a part of a research study in which a number of action selection algorithms have been developed. However, it only provides results obtained with simulated pick-and-place tasks and lacks the application of the presented theories to complex robotic systems.

Another architecture which features special inter-behaviour connections that facilitate the encoding of sequences has been developed at the Computer Science Department of the University of Southern California (USC). As it does not possess a specific name, it will be called “USC behaviour architecture” in the following. The earliest detailed description of the architecture can be found in [Nicolescu 00] and a number of publications describe extensions to the original work. In their papers, the authors identified two limitations of many behaviour-based systems, which they address with their work: BBS are typically not well-suited for realising temporal sequences (1) and lack support for the automatic generation of behaviour networks as well as for the reuse of behaviours (2). The USC behaviour architecture has been implemented using AYLLU, which is an extension of the C language targeting distributed multi-robot behavioural control (see [Werger 00]).

The central elements of the USC behaviour architecture are *abstract* and *primitive* behaviours. Abstract behaviours check the activation conditions of primitive behaviours and activate them in case all conditions are met. There are two types of preconditions: world preconditions, which are related to certain states of the environment, and sequential preconditions, which are related to tasks and allow for specifying complex temporal sequences. By contrast, primitive behaviours realise the actual functionality (like processing sensor data or calculating actuator commands). The authors claim that separating the interface from the actual functionality of a behaviour allows for a more general reuse of the behaviours and for representing a task plan or strategy in the structure of a behaviour network. Figure 2.29 illustrates the difference between a behaviour in a usual architecture and the separation into abstract and primitive behaviours in the USC behaviour architecture.

In Fig. 2.30, the interfaces of the two types of behaviours and their interconnection are depicted. The *UseBehaviour* port can be used to enable or disable the behaviour, while the *Inhibit* port can be used by other behaviours to inhibit it. The *ActivLevel*

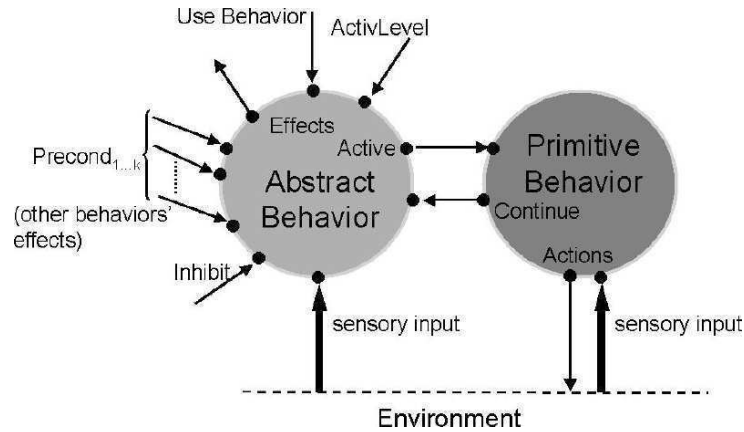


Figure 2.30: The interfaces of an abstract and a primitive behaviour in the USC behaviour architecture and the connections between the two (source: [Nicolescu 02]).

input port can receive so-called activation messages sent out by other behaviours. The activation level of the behaviour is set according to the number of such messages it receives. A behaviour can only get active if its activation level is not 0. Using this technique of activation spreading, behaviours that depend on other behaviours can signal their dependence through the network. This concept of letting behaviours activate or inhibit other behaviours resembles the one described in [Maes 90] (see above). Furthermore, the schemas described in [Dahl 05] exchange messages of activation. The behaviour-based architecture iB2C (see Sec. 2.2.1) also features behaviour activation and inhibition.

The *SensoryInput* ports are used to get information about the environment. Via *Precondition* ports, a behaviour can be informed that other behaviours, on whose execution it depends, have achieved their goals (i.e. that the behaviour's preconditions are fulfilled). This is in turn signalled by the *Effects* output port of a behaviour, which indicates whether the behaviour has achieved its specified effects, i.e. whether the behaviour's so-called postconditions are met. The main connection between an abstract behaviour and its primitive behaviour(s) is established using the *Active* output port of the former and the *Active* input port(s) of the latter. Furthermore, a primitive behaviour can inform its abstract behaviour when its execution is not finished yet using the *Continue* input and output ports. In this case, the abstract behaviour continues to send an activation signal to the corresponding primitive behaviour. Using the *Actions* port, a primitive behaviour can send commands to the robot's actuators.

The USC behaviour architecture features an advantage over the subsumption architecture (see [Nicolescu 02]): It allows for grouping behaviours into a so-called *network abstract behaviour* and thereby abstracting a part of a complete behaviour network into a single component. This resembles the behavioural group in the iB2C (see Sec. 2.2.1). The interface of a network abstract behaviour is the same as the one of a normal abstract behaviour. Hence, a network abstract behaviour can be used within a larger network in place of a standard abstract behaviour. Figure 2.31 depicts a behaviour network in which network abstract behaviours are used to create different levels of abstraction.

Sequences of behaviour execution can be realised in the USC behaviour architecture using the *Effects* output ports and the *Precondition* input ports of abstract behaviours. By

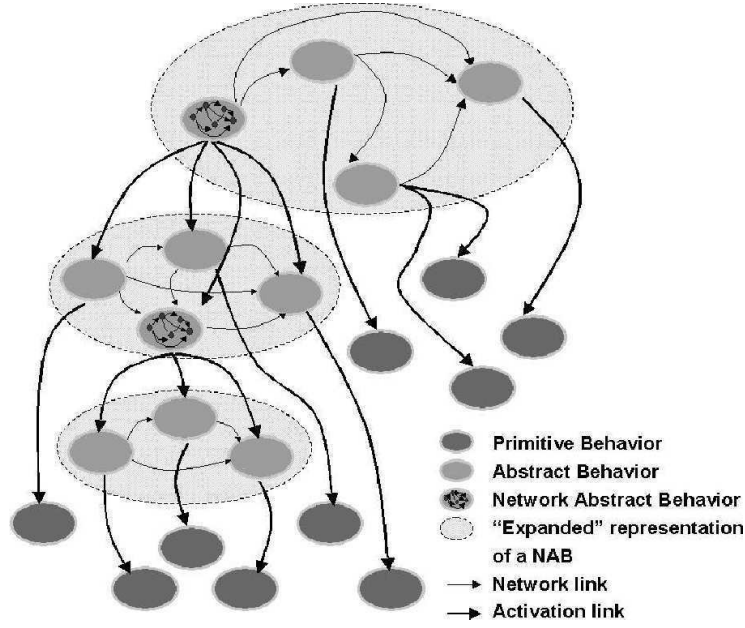


Figure 2.31: A hierarchical behaviour network in which two network abstract behaviours are used to create different levels of abstraction (source: [Nicolescu 02]).

connecting an *Effects* output port of a behaviour B_0 to a *Precondition* input port of a behaviour B_1 , a temporal sequence between the two behaviours is created that depends on the type of precondition that is attributed to the connection. Whether B_1 can get active then depends on the achievement of the goals of behaviour B_0 . There are three such types of preconditions: enabling, ordering, and permanent (see [Nicolescu 01a]). Their meanings are the following:

- **permanent preconditions:** These are preconditions which *must be met during the entire execution* of a behaviour.
- **enabling preconditions:** These are preconditions which *must be met immediately before the activation* of a behaviour.
- **ordering preconditions:** These are preconditions which *must have been met at some point before* the activation of a behaviour.

An example from [Nicolescu 01a] illustrates the differences between the preconditions. The example is employed there to illustrate an algorithm used to learn the structure of behaviour networks from observing the activity of behaviours. It is assumed that the effects of a behaviour A are achieved within the time interval $[t_{1A}, t_{2A}]$ and that a behaviour B is active within the time interval $[t_{1B}, t_{2B}]$. The following three cases correspond to the three preconditions:

Case 1) $t_{1B} \geq t_{1A}$, hence behaviour A is a predecessor of behaviour B . Furthermore, $t_{1B} \leq t_{2A}$ and $t_{2B} \leq t_{2A}$, so the effects of A are permanent preconditions for B .

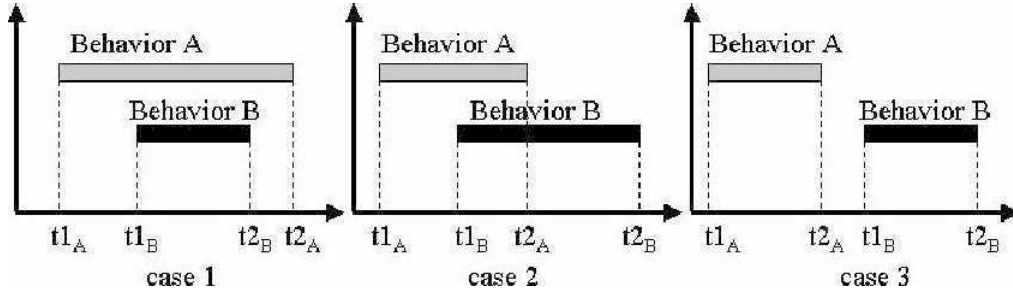


Figure 2.32: An illustration of the three different types of preconditions in the USC behaviour architecture. Case 1 corresponds to a permanent, case 2 to an enabling, and case 3 to an ordering precondition (source: [Nicolescu 01a]).

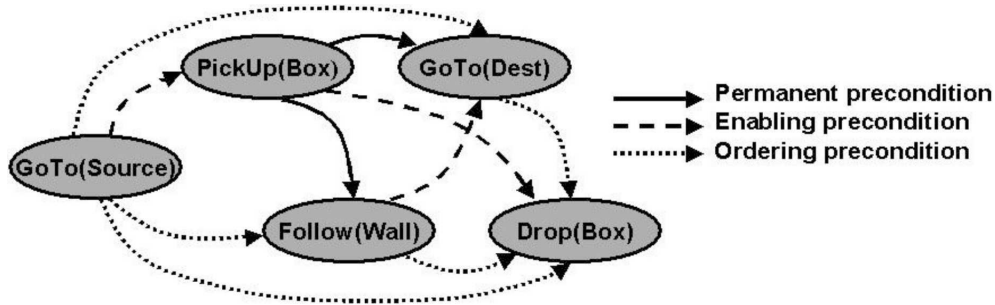


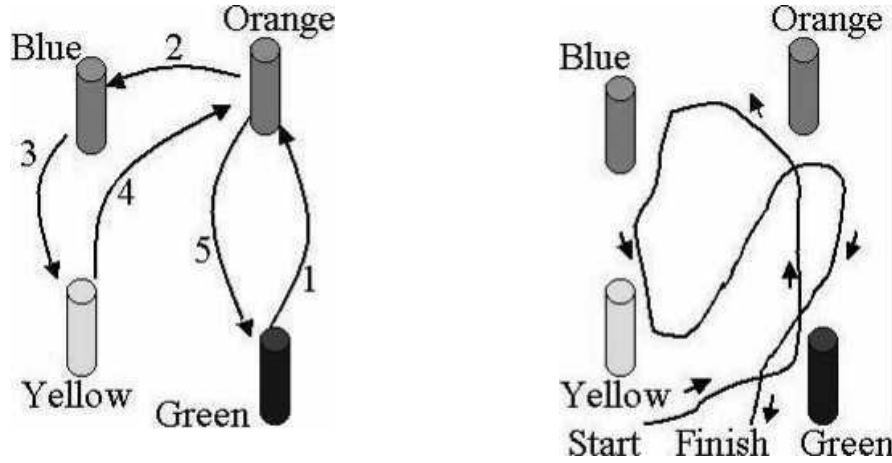
Figure 2.33: A behaviour network created using different types of preconditions (source: [Nicolescu 02]).

Case 2) $t1_B \geq t1_A$, hence behaviour A is a predecessor of behaviour B . Furthermore, $t1_B \leq t2_A$ and $t2_B > t2_A$, so the effects of A are enabling preconditions for B .

Case 3) $t1_B > t2_A$, hence behaviour A is a predecessor of behaviour B and the effects of A are ordering preconditions for B .

Figure 2.32 illustrates these three cases, while Fig. 2.33 depicts a network of behaviours that can be established in the described way.

Using this technique for creating temporal sequences, a robot can learn behaviour networks based on demonstrations by a human or a robot as a teacher (see [Nicolescu 01a] and [Nicolescu 01b]). The robot (a Pioneer 2-DX equipped with two rings of ultrasonic sensors, a SICK laser range finder, a pan-tilt-zoom colour camera, and a gripper) shall learn tasks like visiting a number of coloured targets or moving them from one place to another. Figure 2.34 depicts an experimental setup (a) beside the approximate trajectory of the robot (b). During the (online) demonstration phase, the robot monitors the postconditions of its behaviours and keeps track of whether they are fulfilled or not. For each period of fulfilment, an instance of the corresponding behaviour is created. After that, the robot compares the times when the postconditions started being fulfilled or unfulfilled and from this information creates links between the behaviour instances with the correct types of preconditions. That way, different behaviour networks realising different tasks can be created automatically from a set of available (primitive) behaviours. Another type of human-robot interaction is described in [Nicolescu 03]: In case a robot detects that it is



(a) An experimental setup for learning how to visit a number of coloured targets (source: [Nicolescu 01a]). (b) The approximate trajectory of the robot (source: [Nicolescu 01a]).

Figure 2.34: The images present an experimental setup of a number of coloured targets (see Fig. a) and the approximate trajectory of the robot (see Fig. b).

not capable of executing a task, it searches for a human and tries to express its intentions so that the human will help in executing the task. This mechanism is also realised based on behaviours.

In summary, the USC behaviour architecture allows for building complex behaviour networks by providing different levels of abstraction, offering various types of behaviour activation conditions, and supporting the automatic instantiation as well as connection of behaviours. However, while there is support for creating sequences of behaviour activation, the focus is on learning tasks from demonstration in rather simple, artificial environments. But creating behaviour networks that way is difficult to impossible for real-world applications where complex robot vehicles shall perform sophisticated tasks involving large sequences of behaviour activations. In such cases, demonstration is not the method of choice for defining what a robot shall do. Instead, a—more or less formal—specification of the task (e.g. a finite-state machine) is worked out and then has to be transformed into a control system. The USC behaviour architecture does not help the developer of a behaviour-based system in this process. Furthermore, it does not offer support for any kind of verification. Again, this is important for the use in real-world scenarios as a malfunctioning robot can easily harm people and damage equipment.

2.3 Discussion

In Sec. 2.1, different (formal or graphic) ways of representing (task) sequences have been presented. Their usefulness for the purpose of supporting the developer of a robot control system varies, with more graphic ones being more intuitive and thus easier to use. The graphical representation of Moore machines makes them a good choice for defining behaviour activity sequences, while their textual (and formal) definition allows for automatically processing them in robot control systems. Their selection for defining sequences in the context of this work marks the first design decision:

Design Decision 1

Sequences shall be represented using Moore machines.

From a technical point of view, there are several ways of realising sequences in robot control systems (see Sec. 2.2). They depend heavily on the underlying robot control architecture. Due to the numerous advantages of behaviour-based architectures in general and the many features offered in particular by the iB2C, the latter has been selected for the work at hand. This constitutes the next design decision:

Design Decision 2

The developed concepts for realising sequences shall be integrated into the behaviour architecture iB2C.

Two major techniques of realising sequences in behaviour networks have been presented: centralised (see Sec. 2.2.1) and decentralised (see Sec. 2.2.2). The latter has been identified as advantageous as the actual sequencing logic is not stored in one complex behaviour, but distributed over several behaviours in the system. This facilitates the integration of a behaviour network realising a sequence with other (networks of) behaviours. Furthermore, through the use of special connections like the ones in the USC behaviour architecture, the creation of a network that realises a sequence of tasks can be significantly facilitated. This yields the following design decision:

Design Decision 3

In order to make use of the full spectrum of behaviour-based approaches, the encoding of task sequences within a behaviour network shall be realised in a decentralised way using special inter-behaviour connections.

While the decentralised realisation allows for using the full spectrum of behaviour-based approaches, the manual construction of a network implementing a sequence can easily get complicated. Adequate tool support constitutes a remedy to this problem, which is reflected in the following design decision:

Design Decision 4

To assist the developer when implementing a complex task within an iB2C network, graphical tool support shall be available.

Figure 2.35 shows how the results of this chapter are integrated into the concept of this doctoral thesis.

3. Encoding Task Sequences in iB2C Networks

The previous chapter has presented a number of ways for representing task sequences. Moore automata have been selected for the work described in this dissertation (cp. Design Decision 1) as their graphical representation facilitates their use while their formal definition allows for processing them algorithmically. Using Moore automata, tasks consisting of a sequence of subtasks (e.g. the exploration of an unknown area or the excavation process of an autonomous bucket excavator) can easily be defined.

In the previous chapter, it has been explained that the realisation of such sequences in a robot control system highly depends on the type of the architecture of that system. Several architectures have been mentioned and it has been justified why the behaviour-based approach has been selected for the work at hand. In particular, the behaviour architecture iB2C shall be used as a technical basis for implementing the developed concepts (cp. Design Decision 2).

Two major approaches (centralised and decentralised) to realising sequences in behaviour networks can be distinguished. With the aim of an easy, seamless integration of different components, the decentralised approach has been selected for the work described in this dissertation. It also offers the advantage that no complex component for storing the logic of a sequence has to be created. Moreover, the use of inter-behaviour connections that target specifically at sequences facilitates the creation of the networks in questions (cp. Design Decision 3).

As a technical prerequisite for encoding task sequences in iB2C systems, the realisation of behaviour activity sequences in iB2C networks will be presented in Sec. 3.1. For this purpose, the iB2C is enhanced with a local coordination behaviour offering sequence-specific connections.

The actual encoding of a task sequence that has been defined as a Moore machine in an iB2C network will be presented in Sec. 3.2. Based on the observation that the manual design of such behaviour networks can be tedious and error-prone, a process is suggested in which a Moore machine is first designed using a graphical tool (cp. Design Decision 4)

and then automatically transferred with an algorithm into an iB2C behaviour network encoding the task that is defined by the Moore machine.

3.1 Realising Behaviour Activity Sequences in iB2C Networks

In the preceding chapter, the decentralised encoding of behaviour activity sequences in a behaviour network has been identified as a suitable means for realising sequences and at the same time benefit from the advantages offered by behaviour-based approaches. The already mentioned research done in this field by Nicolescu and Matarić (see Sec. 2.2.2.2) shall serve as a scientific starting point for the work described in the following. In the course of this work, the iB2C (see Sec. 2.2.1) shall be extended with support for the realisation of behaviour activity sequences.

3.1.1 Local Coordination Behaviour

There are different ways in which a behaviour-based architecture can support the encoding of behaviour activity sequences in behaviour networks. Nicolescu and Matarić have chosen to encapsulate the interface of a behaviour with the surrounding network into so-called abstract behaviours in the USC behaviour architecture (see Sec. 2.2.2.2). These are responsible for checking whether a corresponding primitive behaviour is enabled or not as well as whether it is inhibited or not, for calculating whether its goal has been achieved by a corresponding primitive behaviour, and for monitoring the fulfilment of the behaviour's preconditions (see [Nicolescu 00]).

The approach followed in the work at hand differs from the USC approach in that checking to which degree a behaviour is stimulated or inhibited remains part of every single iB2C behaviour—according to the existing definition of behaviours in the iB2C (see Sec. 2.2.1). As this checking only differs in the number of connected inhibiting behaviours, realising it in a behaviour and not an external abstract interface does not complicate the reuse of behaviours in different networks. The processing of the inputs needed for realising sequences, however, is more complex and thus has been sourced out to a local coordination behaviour in order not to increase the complexity of a normal behaviour. This local coordination behaviour is entitled *conditional behaviour stimulator* (CBS) (see [Armbrust 11a]). Each instance of the CBS is responsible for one behaviour working on the achievement of a certain task or for a behaviour group in which a number of behaviours necessary for achieving a certain goal are combined. This approach differs completely from using a complex central component coordinating the sequential execution of several behaviours (like the sequencers described in Sec. 2.2). The encoding of behaviour sequences in iB2C networks results in numerous additional links. By introducing the CBS, a large part of the additional links are connections between different CBS nodes and no additional links are necessary between the normal behaviours. The result is a clearer network structure, in which the majority of inter-behaviour connections only involves CBS nodes and not behaviours executing actual tasks.

The CBS is realised as an iB2C behaviour and thus shares the common interface with all other iB2C behaviours. Its symbol is depicted in Fig. 3.1. The names of CBS nodes are prefixed with “(CBS)” in textual descriptions so that they can be distinguished from other

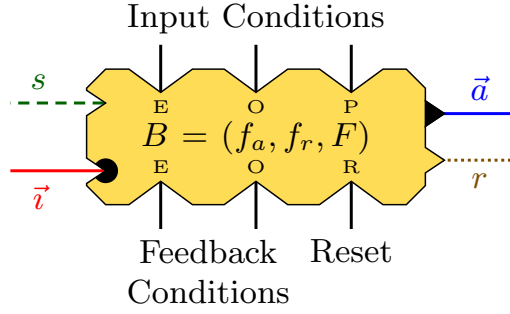


Figure 3.1: The symbol of a CBS depicting the three different types of ports (enabling, ordering, and permanent) for input conditions, the two different types of ports (enabling and ordering) for feedback conditions, and the reset input. As a CBS is a behaviour, it also features the standard behaviour ports.

types of behaviours. In figures, however, this prefix is often left out as the different colours are sufficient for distinguishing different types of behaviours.

A CBS features a number of so-called *input conditions* ic_{CBS} and *feedback conditions* fc_{CBS} . It starts its operation by checking its input conditions. As soon as all of them are fulfilled, the CBS—provided it is activated—gets active, resets its feedback conditions and then starts checking the fulfilment of the latter. At the moment when all feedback conditions are fulfilled, the CBS gets inactive, resets its input conditions, and restarts checking them, i.e. the process starts again. The CBS features a reset input that can be used to set all conditions to unfulfilled so that the CBS gets back to its initial state, in which it restarts checking all of its input conditions¹. Figure 3.2 shows an FSM that illustrates the main phases of the operation of a CBS. For the sake of convenience, the formulae for calculating the fulfilment of conditions, the activity, and the target rating of CBS nodes presented in the remainder of this thesis refer to the phase in which the CBS checks its input conditions. Using the activity of a CBS to stimulate a behaviour that can execute a certain task, it can be achieved that the task is only carried out if certain prerequisites are fulfilled. Such a behaviour shall be called *stimulated behaviour*. As will be shown later (see Sec. 3.2), complex behaviour activity sequences can be created by cascading such constructs.

To each condition, an input port, a threshold, and a type of relation are attributed. In iB2C networks, the coordination between behaviours is mainly established by connecting one behaviour's activity output with another behaviour's stimulation or inhibition input. In the same way, the activity output of a behaviour can be connected to an input port that is attributed to a condition. Furthermore, the second behaviour signal that a behaviour can send out, the target rating, can also be used as an input signal for the input port of a condition. A behaviour that is connected to a port of a CBS associated with an input condition is called *input behaviour*. Correspondingly, a behaviour that is connected to a port of a CBS associated with a feedback condition is called *feedback behaviour*. A result of this type of connections is that the condition input ports of a CBS take values from $[0, 1]$ and the thresholds are chosen from the same interval. The type of relation can be one of $\{<, \leq, =, \geq, >, \neq\}$. As described in [Nicolescu 02], the check whether a behaviour's preconditions are fulfilled in the USC behaviour architecture is based on the check of

¹The reset input was missing in the original version of the CBS presented in [Armbrust 11a].

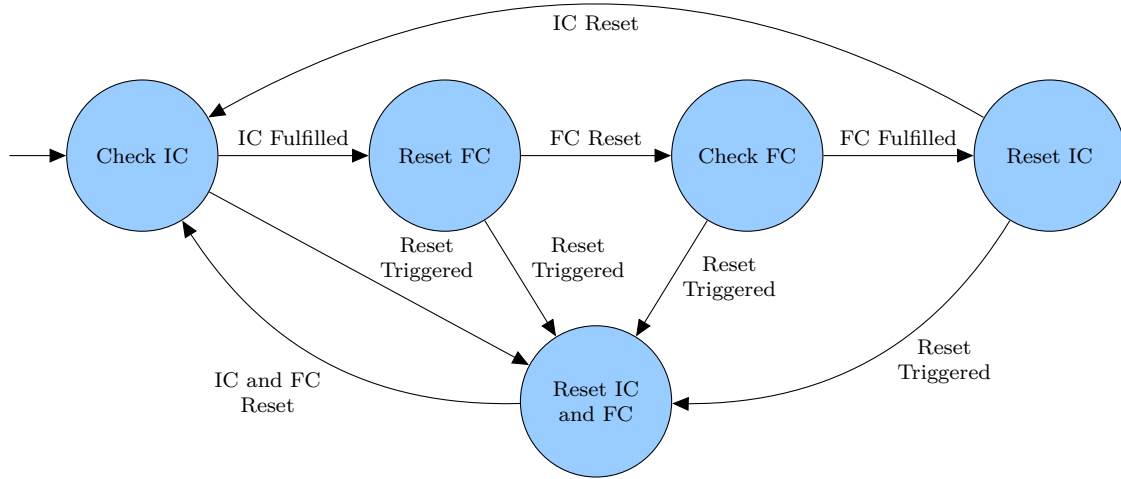


Figure 3.2: An FSM illustrating the main phases of the operation of a CBS (IC: input conditions; FC: feedback conditions). “Reset Triggered” refers to the reset input of the CBS.

a binary value (as the *Effects* output is binary) and is thus limited in contrast to the approach presented here.

For each input condition $(ic_{CBS})_j$ of a CBS, an input relation $(ir_{CBS})_j$ is defined as follows:

$$(ir_{CBS})_j(t) = \begin{cases} 1 & \text{if } (is_{CBS})_j (\overset{i}{\otimes}_{CBS})_j (it_{CBS})_j, \\ 0 & \text{else} \end{cases} \quad (3.1)$$

where t is the time, $(is_{CBS})_j$ is the input signal, $(\overset{i}{\otimes}_{CBS})_j \in \{<, \leq, =, \geq, >, \neq\}$ is the type of the input relation $(ir_{CBS})_j$, $(it_{CBS})_j$ is the input threshold, and $j \in \{1, \dots, n_{ic}\}$ with n_{ic} being the number of input conditions. In the same way, a feedback relation $(fr_{CBS})_j$ with the feedback signal $(fs_{CBS})_j$, the feedback relation $(\overset{f}{\otimes}_{CBS})_j$, and the feedback threshold $(ft_{CBS})_j$ is defined for each feedback condition $(fc_{CBS})_j$.

Whether or not a condition is fulfilled depends on the fulfilment of its relation and on the condition’s type. According to [Nicolescu 01b], three different types of conditions are distinguished here (see also [Armbrust 11a]):

1. **Permanent:** The corresponding relation has to be fulfilled during the whole time when the behaviour shall be active, i.e. the condition is fulfilled if and only if the relation is fulfilled (cp. Eq. 3.2).
2. **Ordering:** The corresponding relation has to be fulfilled at some point in time before the behaviour shall get active. The condition will stay fulfilled independent of whether the relation stays fulfilled or not (cp. Eq. 3.3).
3. **Enabling:** The corresponding relation has to be fulfilled at the exact point in time when the behaviour shall get active. After that, the condition stays fulfilled independent of the fulfilment of the relation (cp. Eq. 3.4).

This can be expressed in a more formal way as described in the following for a CBS with n_{IC} input conditions.

Permanent

$$(ic_{CBS})_j(t) = \begin{cases} 1 & \text{if } (ir_{CBS})_j(t) = 1 \\ 0 & \text{else} \end{cases} \quad (3.2)$$

The check whether a permanent condition is fulfilled or not is the simplest of the three conditions: A permanent condition is fulfilled at time t if and only if the corresponding relation is fulfilled at time t .

Ordering

$$(ic_{CBS})_j(t) = \begin{cases} 1 & \text{if } \exists t_0 \leq t : (ir_{CBS})_j(t_0) = 1 \\ 0 & \text{else} \end{cases} \quad (3.3)$$

For checking the fulfilment of an ordering condition, it is not sufficient to check the fulfilment of the corresponding input relation at the current time as a fulfilment of the input relation in the past is sufficient for the condition to be fulfilled at the current time. In more formal terms, an ordering condition is fulfilled at time t if there is a point in time $t_0 \leq t$ at which the corresponding relation was fulfilled. No assumption is made about the fulfilment of the relation after this moment t_0 .

Enabling

$$(ic_{CBS})_j(t) = \begin{cases} 1 & \text{if } \exists t_0 \leq t : \\ & \left(\bigwedge_{k=1}^{n_{IC}} (ir_{CBS})_k(t_0) = 1 \right)_{(ic_{CBS})_k \text{ enabling}} \\ & \wedge \left(\bigwedge_{k=1}^{n_{IC}} (ic_{CBS})_k(t_0) = 1 \right)_{(ic_{CBS})_k \text{ ordering}} \\ & \wedge \left(\bigwedge_{k=1}^{n_{IC}} (ic_{CBS})_k(t_1) = 1 \quad \forall t_1 : t_0 \leq t_1 \leq t \right)_{(ic_{CBS})_k \text{ permanent}} \\ 0 & \text{else} \end{cases} \quad (3.4)$$

The check whether an enabling condition is fulfilled or not is the most complex one. In order to determine whether an enabling condition is fulfilled at time t , it has to be checked whether there is a point in time $t_0 \leq t$ at which all conditions were fulfilled for the first time and since which all conditions have been fulfilled. For the ordering conditions, this

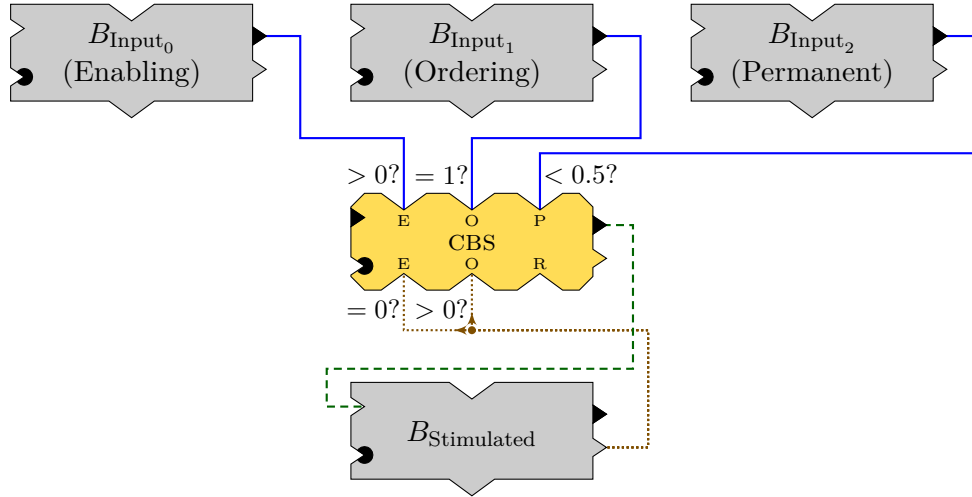


Figure 3.3: A simple network consisting of a CBS and four standard behaviours. Three of the latter are input behaviours that are connected to ports corresponding to enabling, ordering, and permanent input conditions of the CBS. The remaining behaviour is stimulated by the CBS and is also connected to ports corresponding to enabling and ordering feedback conditions.

simply means that it has to be checked whether they were fulfilled at t_0 . If yes, they are still fulfilled at t . For permanent conditions, it is necessary to check their fulfilment for all t_1 with $t_0 \leq t_1 \leq t$. For enabling conditions, a cyclic dependency in the checks would occur if the fulfilment of one enabling condition depended on the fulfilment of another and vice versa. Hence, the fulfilment of the corresponding relations instead of the conditions is checked for them.

There are enabling, ordering, and permanent input conditions, but only enabling and ordering feedback conditions². As a CBS stops checking its feedback conditions as soon as all of them are fulfilled and restarts checking its input conditions, there would be no difference between enabling and permanent feedback conditions.

In case a CBS is in the process of monitoring its input conditions, its activity a_{CBS} and target rating r_{CBS} are calculated as follows:

$$a_{CBS}(t) = s_{CBS}(t) \cdot (1 - i_{CBS}(t)) \cdot \prod_{j=1}^{n_{ic}} (ic_{CBS})_j(t) = \iota_{CBS} \cdot \prod_{j=1}^{n_{ic}} (ic_{CBS})_j(t) \quad (3.5)$$

$$r_{CBS}(t) = \prod_{j=1}^{n_{ic}} (ic_{CBS})_j(t) \quad (3.6)$$

Figure 3.3 depicts a small behaviour network that shall be used to illustrate the functionality of the CBS. The network consists of a CBS and four standard behaviours. Three of the latter (B_{Input_0} , B_{Input_1} , and B_{Input_2}) are connected to the CBS as input behaviours using their activity output ports and input ports of the CBS that correspond to enabling, ordering,

²In contrast to this, the original version of the CBS presented in [Armbrust 11a] featured permanent feedback conditions. The symbol of the CBS depicted in Fig. 3.1 shows the reset port in the former place of the port for permanent feedback conditions.

and permanent input conditions, respectively. The corresponding thresholds and types of relation are denoted besides the input ports. The indices j_{Input} of the input relations $(\text{ir}_{\text{CBS}})_{j_{\text{Input}}}$ and the input conditions $(\text{ic}_{\text{CBS}})_{j_{\text{Input}}}$ with $0 \leq j_{\text{Input}} \leq 2$ refer to the enabling, ordering, and permanent input relations and conditions, respectively, in this order. The following formulae describe the input relations:

$$(\text{ir}_{\text{CBS}})_0(t) = \begin{cases} 1 & \text{if } a_{\text{Input}_0}(t) > 0 \\ 0 & \text{else} \end{cases} \quad (\text{ir}_{\text{CBS}})_1(t) = \begin{cases} 1 & \text{if } a_{\text{Input}_1}(t) = 1 \\ 0 & \text{else} \end{cases}$$

$$(\text{ir}_{\text{CBS}})_2(t) = \begin{cases} 1 & \text{if } a_{\text{Input}_2}(t) < 0.5 \\ 0 & \text{else} \end{cases}$$

The input conditions can be described as follows:

$$(\text{ic}_{\text{CBS}})_0(t) = \begin{cases} 1 & \text{if } \exists t_0 \leq t : \begin{aligned} & \left(a_{\text{Input}_0}(t_0) > 0 \right) \\ & \wedge \left(\exists t_1 \leq t_0 : a_{\text{Input}_1}(t_1) = 1 \right) \\ & \wedge \left(a_{\text{Input}_2}(t_1) < 0.5 \quad \forall t_1 : t_0 \leq t_1 \leq t \right) \end{aligned} \\ 0 & \text{else} \end{cases}$$

$$(\text{ic}_{\text{CBS}})_1(t) = \begin{cases} 1 & \text{if } \exists t_0 \leq t : a_{\text{Input}_1}(t_0) = 1 \\ 0 & \text{else} \end{cases} \quad (\text{ic}_{\text{CBS}})_2(t) = \begin{cases} 1 & \text{if } a_{\text{Input}_2}(t) < 0.5 \\ 0 & \text{else} \end{cases}$$

In other words, the input conditions are fulfilled from the point in time t_0 on at which the following holds:

1. The activity of B_{Input_0} is > 0 , i.e. $a_{\text{Input}_0}(t_0) > 0$.
2. The activity of B_{Input_1} was (or still is) 1, i.e. there is a $t_1 \leq t_0$ such that $a_{\text{Input}_1}(t_1) = 1$.
3. The activity of B_{Input_2} is < 0.5 , i.e. $a_{\text{Input}_2}(t_0) < 0.5$.

From this time on, the CBS will stay active as long as $a_{\text{Input}_2} < 0.5$ and the feedback conditions are not fulfilled yet.

Three connections are used to send signals from the CBS to the stimulated behaviour $B_{\text{Stimulated}}$ and vice versa: The activity output of the CBS is connected to the stimulating input of the stimulated behaviour. This results in a stimulation of the latter as soon as the activity of the CBS rises.

Two further connections have been drawn from the target rating output of the stimulated behaviour to two inputs of the CBS associated with an enabling and an ordering feedback condition, respectively. Hence, $B_{\text{Stimulated}}$ is not only a stimulated behaviour with respect

to the CBS, but also a feedback behaviour. The indices j_{Feedback} of the feedback relations $(\text{fr}_{\text{CBS}})_{j_{\text{Feedback}}}$ and the feedback conditions $(\text{fc}_{\text{CBS}})_{j_{\text{Feedback}}}$ with $0 \leq j_{\text{Feedback}} \leq 1$ are defined in the same way as those of the input relations and conditions, i.e. they refer to the enabling and ordering feedback relations and conditions, respectively, in this order. The feedback relations can then be described as follows:

$$(\text{fr}_{\text{CBS}})_0(t) = \begin{cases} 1 & \text{if } r_{\text{Stimulated}}(t) = 0 \\ 0 & \text{else} \end{cases} \quad (\text{fr}_{\text{CBS}})_1(t) = \begin{cases} 1 & \text{if } r_{\text{Stimulated}}(t) > 0 \\ 0 & \text{else} \end{cases}$$

The feedback conditions are defined by the following two formulae:

$$(\text{fc}_{\text{CBS}})_0(t) = \begin{cases} 1 & \text{if } \exists t_0 \leq t : (r_{\text{Stimulated}}(t_0) = 0) \\ & \wedge (\exists t_1 \leq t_0 : a_{\text{Stimulated}}(t) > 0) \\ 0 & \text{else} \end{cases}$$

$$(\text{fc}_{\text{CBS}})_1(t) = \begin{cases} 1 & \text{if } \exists t_0 \leq t : r_{\text{Stimulated}}(t) > 0 \\ 0 & \text{else} \end{cases}$$

The combination of the five behaviours in the described way results in the following overall behaviour of the network: If the stimulated behaviour $B_{\text{Stimulated}}$ is not completely satisfied with the current situation (i.e. $r_{\text{Stimulated}} > 0$), it will get active (i.e. $a_{\text{Stimulated}} > 0$) and start executing its task as soon as it gets stimulated by the CBS—in case it is not inhibited. In the context of this example, no maximum duration of the task is specified. Hence, the execution of the task can take an arbitrary amount of time. As long as it is activated and not yet fully satisfied with the situation, $B_{\text{Stimulated}}$ tries to alter the situation in a way that increases its satisfaction so that it finally achieves full satisfaction, whereupon its target rating goes back down to 0. Due to the two feedback conditions, the rise and fall of $r_{\text{Stimulated}}$ signals the CBS that $B_{\text{Stimulated}}$ has completed its job and does not need to be stimulated any longer. While this can be considered the intended way of how the behaviours of this network interact, it is not the only possible scenario. For example, it is possible that the permanent input condition stops being fulfilled, which would result in the CBS getting inactive. This in turn would result in $B_{\text{Stimulated}}$ losing its stimulation, which means that its activity would drop to 0 and the behaviour would lose its influence in the network. Furthermore, it is possible that the CBS or $B_{\text{Stimulated}}$ are inhibited by another behaviour, which would also result in $B_{\text{Stimulated}}$ not being able to fulfil its task. Even if $B_{\text{Stimulated}}$ can stay active, it is not guaranteed that it is able to achieve its goal, resulting in its target rating staying above 0. This kind of situation is not unusual in a complex system and can be dealt with, for example, by a behaviour that monitors the situation and takes action if no progress is made. Such an action could be inhibiting the CBS or $B_{\text{Stimulated}}$ and giving control to another sub-network.

Figures 3.4 and 3.5 depict a possible sequence of input and feedback signals along with the values of the corresponding input relations and feedback relations as well as input

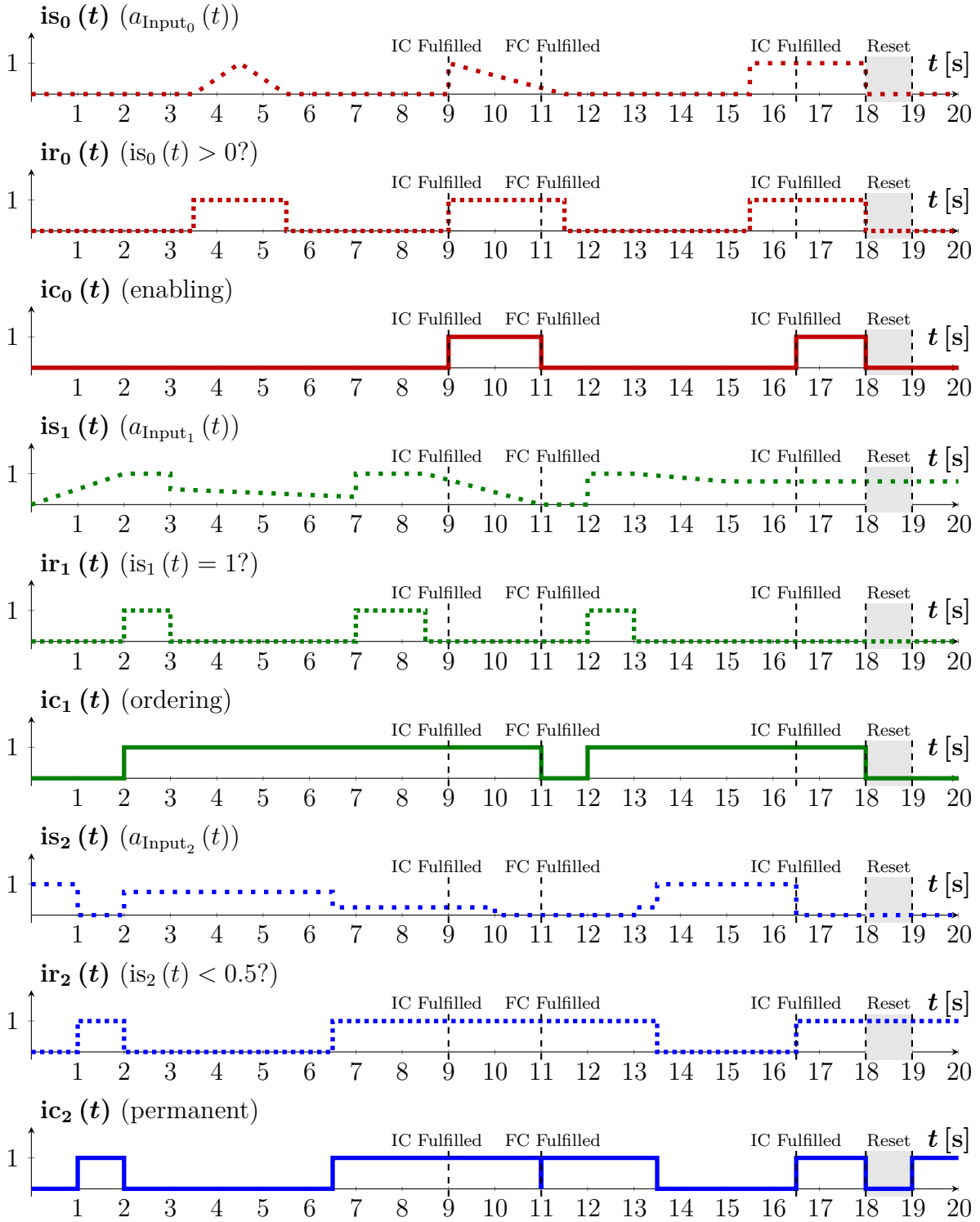


Figure 3.4: An example sequence of the values of a CBS. The red graphs depict $is_0(t)$, $ir_0(t)$, and $ic_0(t)$, the green graphs depict $is_1(t)$, $ir_1(t)$, and $ic_1(t)$, and the blue graphs depict $is_2(t)$, $ir_2(t)$, and $ic_2(t)$, for $t : 0s \leq t \leq 20s$.

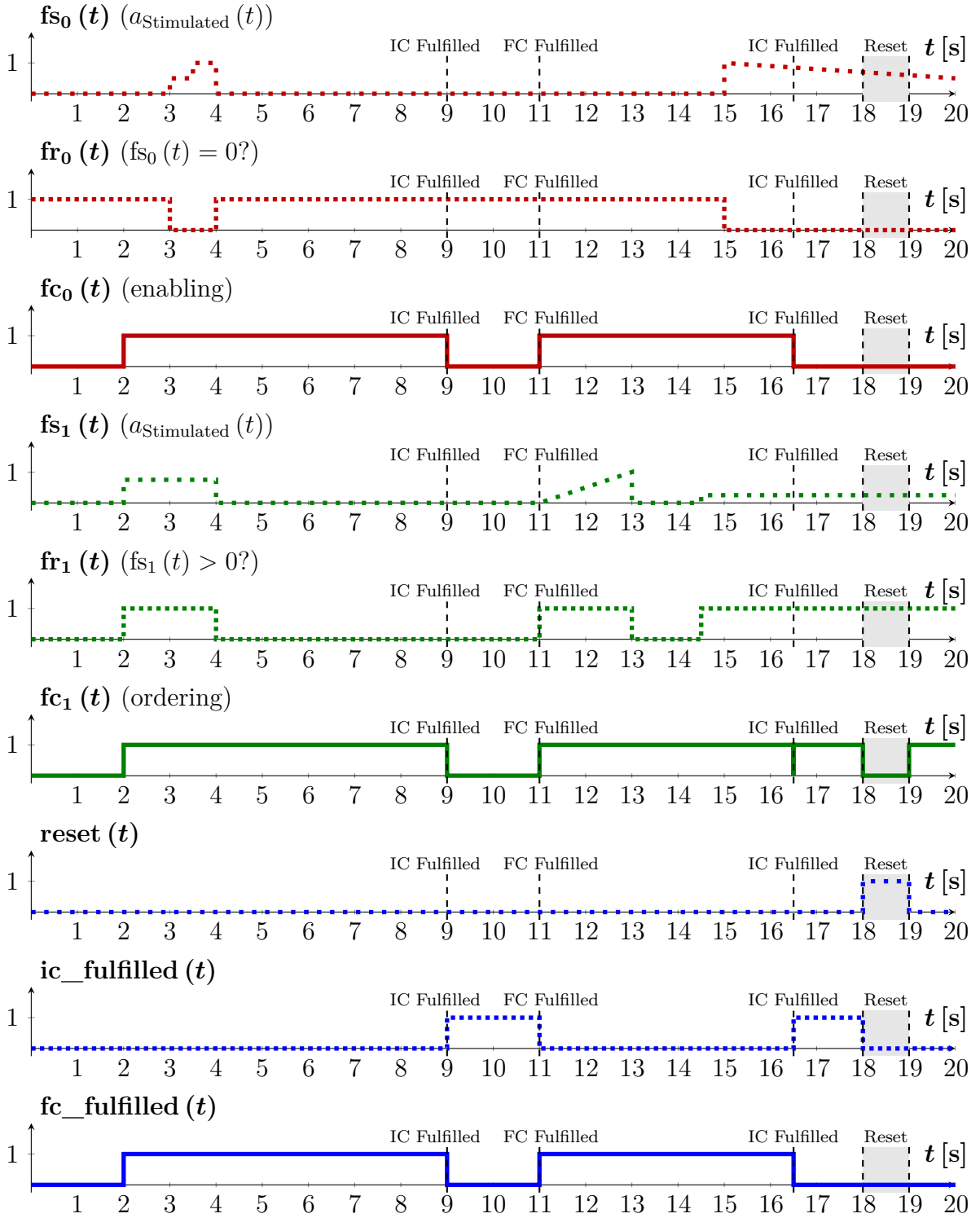


Figure 3.5: An example sequence of the values of a CBS. The red graphs depict $fs_0(t)$, $fr_0(t)$, and $fc_0(t)$ and the green graphs depict $fs_1(t)$, $fr_1(t)$, and $fc_1(t)$. The blue graphs depict the reset signal $reset(t)$, the fulfilment of all input conditions $ic_fulfilled(t)$, and the fulfilment of all feedback conditions $fc_fulfilled(t)$.

and feedback conditions of the CBS in the example network (see Fig. 3.3). In Fig. 3.4, the graphs of the input signals is_0 , is_1 , and is_2 , of the input relations ir_0 , ir_1 , and ir_2 , and of the input conditions ic_0 , ic_1 , and ic_2 are depicted. Fig. 3.5 shows the graphs of the feedback signals fs_0 and fs_1 , of the feedback relations fr_0 and fr_1 , of the feedback conditions fc_0 and fc_1 , of the external reset signal, and of the functions $ic_fulfilled$ and $fc_fulfilled$, which indicate whether all input conditions and all feedback conditions, respectively, are fulfilled.

The first input signal $is_0(t)$ starts rising from $t = 3.5$ s on, resulting in $ir_0(t)$ becoming 1. However, the other two input conditions are not fulfilled until after $is_0(t)$ falls back to 0, hence $ic_0(t)$ stays 0 until $t = 9$ s. ir_1 is fulfilled at $t = 2$ s for the first time, because $is_1(t) < 1 \forall t < 2$ s. As ic_1 is an ordering condition, $ic_1(t) = 1 \forall t : 2 \text{ s} \leq t < 11$ s. ic_2 is a permanent condition. Therefore, it is fulfilled if and only if $ir_2(t) = 1$, i.e. if and only if $is_2(t) < 0.5$. Hence, the graphs of $ir_2(t)$ and $ic_2(t)$ are equal. $fr_0(t) = 1$ for $t \leq 15$ s except for $t : 3 \text{ s} \leq t < 4$ s. Furthermore, $fc_1(t) = 1$ from $t = 2$ s on. As a result, $fc_0(t) = 1$ from $t = 2$ s to $t = 9$ s. At $t = 9$ s, all three input conditions are fulfilled simultaneously for the first time, triggering a reset of the feedback condition. Thus, $t = 9$ s corresponds to t_0 in Eq. 3.4. The CBS then switches to checking the fulfilment of the feedback conditions. Only 2 s later, the enabling and the ordering feedback conditions are fulfilled. This in turn triggers a reset of the input conditions and a restart of checking the fulfilment of the input conditions. The permanent input condition is directly fulfilled, shortly followed by the ordering input condition, which is fulfilled from $t = 12$ s on. As $ir_0(t)$ goes down to 0 at $t = 11.5$ s, not all input conditions are fulfilled. $ir_0(t)$ rises back to 1 at $t = 15.5$ s, but due to $ir_2(t) = 0$ from $t = 13.5$ s on, still not all input conditions are fulfilled. Finally, $ir_2(t)$ goes up to 1 at $t = 16.5$ s, which results in all input conditions being fulfilled again and thus starts the second phase in which the fulfilment of the feedback conditions is checked. Shortly after that, the reset input is 1 (from $t = 18$ s to $t = 19$ s), resulting in all conditions being reset. Hence, the fulfilment of the input conditions has to be checked again.

The work of Nicolescu and Matarić (cp. Sec. 2.2.2.2) has shown that the concept of having enabling, ordering, and permanent conditions that must be fulfilled before a behaviour can get active is sound and allows for creating networks of behaviour activity sequences. Based on the behaviour network depicted in Fig. 3.6, it shall be illustrated how different combinations of input and feedback conditions can allow for different temporal sequences of the activities of two behaviours. The network consists of two standard behaviours B_0 and B_1 as well as a CBS. B_0 is permanently stimulated (indicated by the black triangle in its stimulation port). Its activity output is connected to ports of the CBS that are associated with input and feedback conditions, respectively. The CBS is also permanently stimulated. It uses its activity a_{CBS} to stimulate B_1 . The activity of B_1 can rise to 1 as soon as $\iota_{B_1} = 1$. When exactly this happens depends on the internal calculation of a_{B_1} in B_1 and cannot be determined directly by the CBS. In other words, the CBS determines the earliest point in time at which B_1 can get active as well as the latest point in time at which it has to get inactive again. No behaviour is connected to the inhibition port of B_1 in order to avoid possible influences of the inhibiting behaviour. The three input relations (enabling, ordering, and permanent) shall be named $(ir_{CBS})_E$, $(ir_{CBS})_O$, and $(ir_{CBS})_P$ and the corresponding input conditions $(ic_{CBS})_E$, $(ic_{CBS})_O$, and $(ic_{CBS})_P$. Accordingly, the two feedback relations (enabling and ordering) shall be named $(fr_{CBS})_E$ and $(fr_{CBS})_O$ and the corresponding feedback conditions $(fc_{CBS})_E$ and $(fc_{CBS})_O$.

Figure 3.7 presents several graphs that give an overview of possible temporal sequences

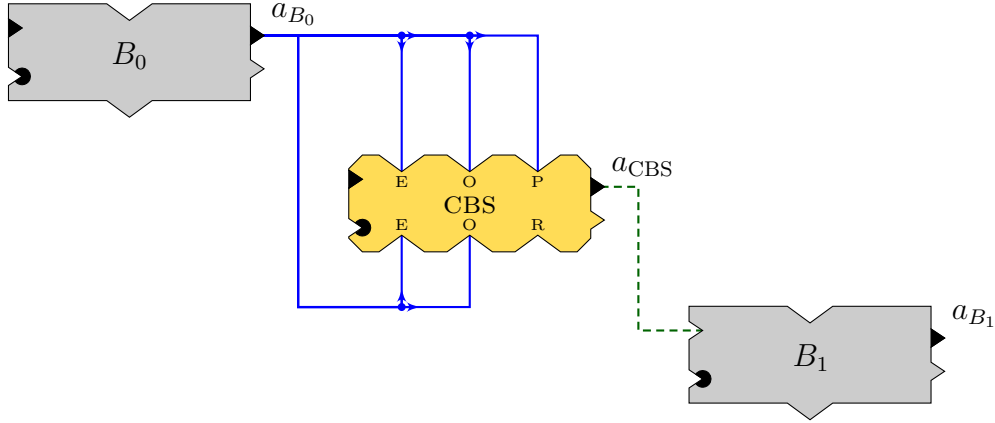


Figure 3.6: A behaviour network consisting of three behaviours: two standard behaviours B_0 and B_1 as well as a CBS. B_0 serves as an input behaviour and as a feedback behaviour for the CBS using its activity a_{B_0} . The CBS stimulates B_1 with its activity a_{CBS} . A black triangle in a behaviour's stimulation port means that the behaviour is always stimulated. Thus, B_0 and the CBS are constantly stimulated.

in which the changes of the activities of the two behaviours B_0 and B_1 can occur. The horizontal axes of the graphs show the time t , while the vertical axes show the activity a_{B_1} of the behaviour B_1 (Graph 0)) and the activity a_{B_0} of the behaviour B_0 (Graphs 1) to 24)), respectively. Graph 0) depicts that a_{B_1} is 0 up to time t_{1R} , when it rises to 1. It stays there until time t_{1F} and then falls back to 0 again. Graphs 1) to 24) depict different points in time at which a_{B_0} rises from 0 to 1 (t_{0R}) and falls back to 0 (t_{0F}), respectively. The extreme cases in which $t_{0R} = -\infty$ or $t_{0F} = +\infty$, i.e. a_{B_0} is already 1 or does not go back to 0, are also possible. In all cases, there is at most one rise and one fall of a_{B_0} and exactly one rise and one fall of a_{B_1} . The red dashed lines indicate t_{1R} and t_{1F} . The temporal sequence of t_{0R} , t_{0F} , t_{1R} , and t_{1F} varies depending on t_{0R} and t_{0F} . In total, 24 cases can be distinguished that way. There is only one limitation: $t_{0R} < t_{0F}$, i.e. the activity of B_0 first rises and then falls. The following cases are depicted in Fig. 3.7. Their numbers refer to the numbers of the graphs.

- | | |
|--|--|
| 1) $-\infty = t_{0R} < t_{0F} < t_{1R} < t_{1F} < \infty$ | 13) $-\infty < t_{0R} = t_{1R} < t_{0F} < t_{1F} < \infty$ |
| 2) $-\infty = t_{0R} < t_{0F} = t_{1R} < t_{1F} < \infty$ | 14) $-\infty < t_{0R} = t_{1R} < t_{0F} = t_{1F} < \infty$ |
| 3) $-\infty = t_{0R} < t_{1R} < t_{0F} < t_{1F} < \infty$ | 15) $-\infty < t_{0R} = t_{1R} < t_{1F} < t_{0F} < \infty$ |
| 4) $-\infty = t_{0R} < t_{1R} < t_{0F} = t_{1F} < \infty$ | 16) $-\infty < t_{0R} = t_{1R} < t_{1F} < t_{0F} = \infty$ |
| 5) $-\infty = t_{0R} < t_{1R} < t_{1F} < t_{0F} < \infty$ | 17) $-\infty < t_{1R} < t_{0R} < t_{0F} < t_{1F} < \infty$ |
| 6) $-\infty = t_{0R} < t_{1R} < t_{1F} < t_{0F} = \infty$ | 18) $-\infty < t_{1R} < t_{0R} < t_{0F} = t_{1F} < \infty$ |
| 7) $-\infty < t_{0R} < t_{0F} < t_{1R} < t_{1F} < \infty$ | 19) $-\infty < t_{1R} < t_{0R} < t_{1F} < t_{0F} < \infty$ |
| 8) $-\infty < t_{0R} < t_{0F} = t_{1R} < t_{1F} < \infty$ | 20) $-\infty < t_{1R} < t_{0R} < t_{1F} < t_{0F} = \infty$ |
| 9) $-\infty < t_{0R} < t_{1R} < t_{0F} < t_{1F} < \infty$ | 21) $-\infty < t_{1R} < t_{0R} = t_{1F} < t_{0F} < \infty$ |
| 10) $-\infty < t_{0R} < t_{1R} < t_{0F} = t_{1F} < \infty$ | 22) $-\infty < t_{1R} < t_{0R} = t_{1F} < t_{0F} = \infty$ |
| 11) $-\infty < t_{0R} < t_{1R} < t_{1F} < t_{0F} < \infty$ | 23) $-\infty < t_{1R} < t_{1F} < t_{0R} < t_{0F} < \infty$ |
| 12) $-\infty < t_{0R} < t_{1R} < t_{1F} < t_{0F} = \infty$ | 24) $-\infty < t_{1R} < t_{1F} < t_{0R} < t_{0F} = \infty$ |

As can be seen, t_{0R} is first kept fix and t_{0F} is moved through the different sections. Then t_{0R} is advanced to the next section and t_{0F} is again moved through the remaining sections—and so on. The question to answer is which combinations of input and feedback conditions allow for the activities of B_0 and B_1 to produce the temporal sequences depicted in the graphs of Fig. 3.7.

Table 3.1 provides possible definitions for the input and feedback relations so that they allow for the temporal sequences of a_{B_0} and a_{B_1} rising and falling as depicted in Fig. 3.7. It shall be mentioned that when an activity changes at exactly t_0 , the new value of the activity (i.e. *after* the change) is taken into account. The table can be read as follows: The temporal sequence of t_{0R} , t_{0F} , t_{1R} , and t_{1F} corresponding to the number in the first column is made possible by setting the five conditions of the CBS in the behaviour network depicted in Fig. 3.6 according to the remaining columns of the table. A “-” in a column of a condition means that no condition of that type can be defined for the sequence in question. The last column attributes a unique ID to all combinations of conditions, i.e. cases that are enabled by the same combination of conditions also have the same ID. For example, the sequence shown in Graph 7) is made possible with the following input and feedback relations:

$$\begin{aligned}
 (\text{irCBS})_{\text{E}}(t) &= \begin{cases} 1 & \text{if } a_{B_0} = 0 \\ 0 & \text{else} \end{cases} & (\text{irCBS})_{\text{O}}(t) &= \begin{cases} 1 & \text{if } a_{B_0} = 1 \\ 0 & \text{else} \end{cases} \\
 (\text{irCBS})_{\text{P}}(t) &= \begin{cases} 1 & \text{if } a_{B_0} = 0 \\ 0 & \text{else} \end{cases} & (\text{frCBS})_{\text{E}}(t) &= \begin{cases} 1 & \text{if } a_{B_0} = 1 \\ 0 & \text{else} \end{cases}
 \end{aligned}$$

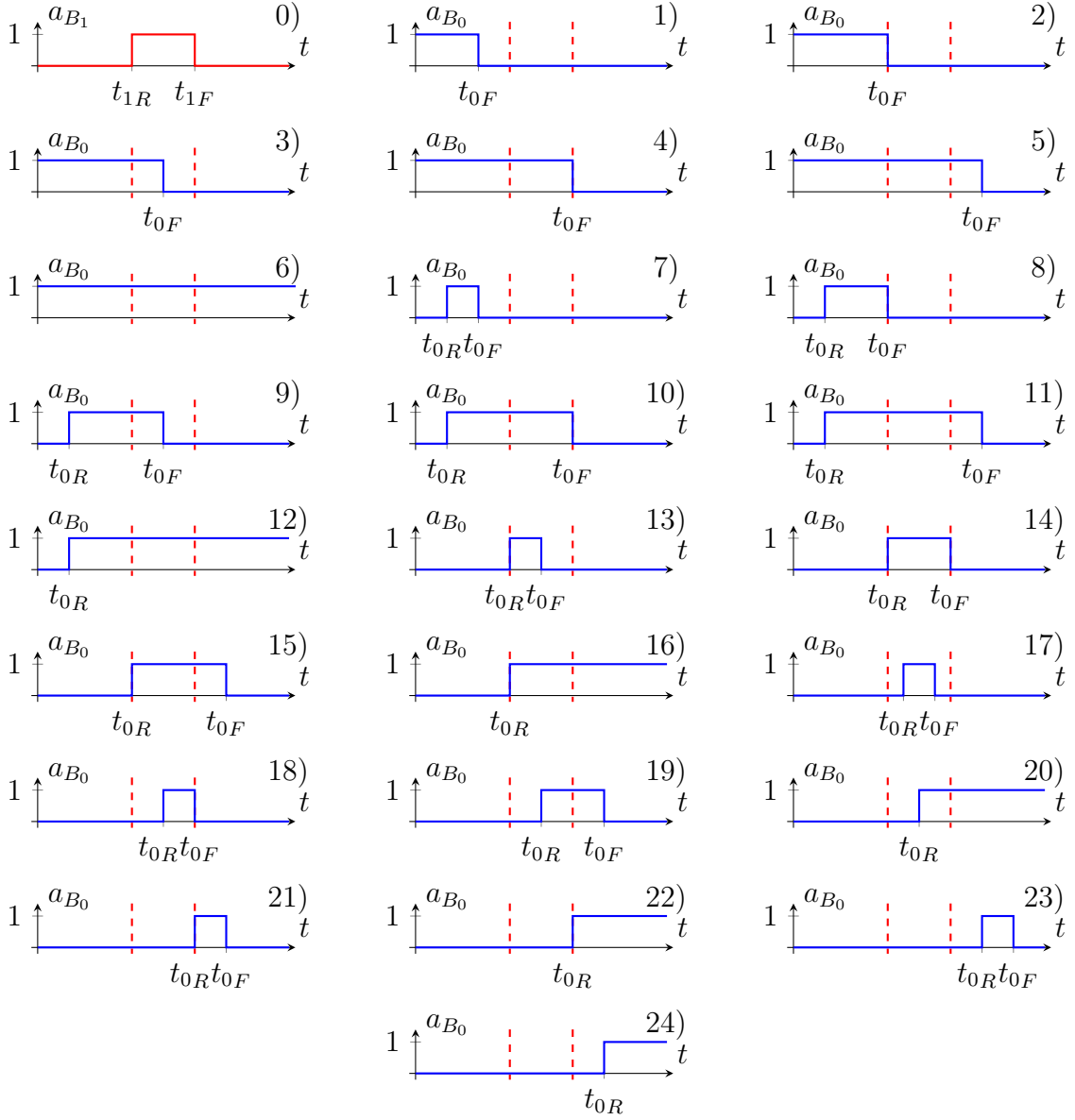


Figure 3.7: Graph 0) depicts the activity of B_1 (a_{B_1}), which rises from 0 to 1 at t_{1R} and falls back to 0 at t_{1F} . Graphs 1) to 24) depict the activity of B_0 (a_{B_0}), which rises from 0 to 1 at t_{0R} and falls back to 0 at t_{0F} . The red dashed lines indicate the points t_{1R} and t_{1F} , respectively. Thereby, different temporal sequences of t_{0R} , t_{0F} , t_{1R} , and t_{1F} are illustrated.

$$(\text{fr}_{\text{CBS}})_O(t) = \begin{cases} 1 & \text{if } a_{B_0} = 0 \\ 0 & \text{else} \end{cases}$$

The unique IDs in the last column of the table show that each of the listed combinations of input and feedback conditions (with the exception of Case 6)) has been assigned to more than one temporal sequence of the activities a_0 and a_1 . For example, the sequence depicted in Graph 8) is possible with the combination of conditions that also allows for the sequence depicted in Graph 7). The reason why this is possible is that only the order of events is taken into account, not the timespan between them. The CBS only allows for B_1 to become active by stimulating it, but it does not force it to get active, i.e. it is possible that B_1 is stimulated but does not get active at once. For example, the input conditions are fulfilled at t_{0F} in Case 7), but $t_{0F} < t_{1R}$, i.e. B_1 does not get active directly when the CBS starts stimulating it. By contrast, it does so in Case 8). The table shows that a simple behaviour network as the one depicted in Fig. 3.6 can allow for a large number of temporal sequences of the activities of two involved behaviours. With a direct stimulating connection from B_0 to B_1 (i.e. without the intermediary CBS), fewer sequences would be possible. This demonstrates the value that the CBS adds to the iB2C.

In the above example, one behaviour (B_1) has two roles with respect to the CBS: It constitutes an input behaviour as well as a feedback behaviour. This is not necessary: As has been depicted in Fig. 3.3, the stimulated behaviour can also be the feedback behaviour. By employing more than one CBS in a network, more complex temporal behaviour activity sequences can be realised, demonstrating the powerfulness of the concept chosen for the realisation of the CBS node. In the following, two sub-networks from the control system of the off-road vehicle RAVON which realise behaviour activity sequences will be presented. Section 3.2 will show how to handle the complexity of larger networks containing several CBS nodes by explaining how a high-level task consisting of a number of subtasks can be modelled as a finite-state machine and then translated according to fixed rules into a corresponding behaviour network.

3.1.2 Example Applications

After the formal aspects of the local coordination behaviour CBS have been described in the previous section, its application to real-world problems shall be presented in the following. Two examples will be given that show how the CBS can be used to encode sequences in behaviour networks.

Both examples originate from the control system of the autonomous off-road robot RAVON (see Fig. 2.20). This robot has been developed as an experimental platform for research concerning hazard detection, environment representation, and navigation in harsh off-road environments. Extensive information about the platform can be found in [Braun 09] (cost-efficient global navigation), [Proetzsch 10] (behaviour-based control system), [Schäfer 11] and [Schäfer 13] (design schemata for the representation, translation, and fusion of environmental information), [Armbrust 09b] (navigation using passages), [Armbrust 10b] (integration of tele-operated, semi autonomous, and fully autonomous control modes using a behaviour-based approach), as well as [Armbrust 11c] (behaviour-based navigation).

Table 3.1: The table shows the enabling, ordering, and permanent input relations as well as the enabling and ordering feedback relations associated with the graphs depicted in Fig. 3.7. a_{B_0} is the activity of behaviour B_0 .

Case	$(\text{ir}_{\text{CBS}})_{\text{E}}$	$(\text{ir}_{\text{CBS}})_{\text{O}}$	$(\text{ir}_{\text{CBS}})_{\text{P}}$	$(\text{fr}_{\text{CBS}})_{\text{E}}$	$(\text{fr}_{\text{CBS}})_{\text{O}}$	ID
1)	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	1
2)	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	1
3)	$a_{B_0} = 1$	$a_{B_0} = 1$	-	$a_{B_0} = 1$	$a_{B_0} = 0$	2
4)	$a_{B_0} = 1$	$a_{B_0} = 1$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	3
5)	$a_{B_0} = 1$	$a_{B_0} = 1$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	3
6)	$a_{B_0} = 1$	$a_{B_0} = 1$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	3
7)	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	1
8)	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	1
9)	$a_{B_0} = 1$	$a_{B_0} = 0$	-	$a_{B_0} = 1$	$a_{B_0} = 0$	4
10)	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	5
11)	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	5
12)	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	5
13)	$a_{B_0} = 1$	$a_{B_0} = 0$	-	$a_{B_0} = 1$	$a_{B_0} = 0$	4
14)	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	5
15)	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	5
16)	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	$a_{B_0} = 1$	5
17)	$a_{B_0} = 0$	$a_{B_0} = 0$	-	-	-	6
18)	$a_{B_0} = 0$	$a_{B_0} = 0$	-	$a_{B_0} = 0$	$a_{B_0} = 1$	7
19)	$a_{B_0} = 0$	$a_{B_0} = 0$	-	$a_{B_0} = 0$	$a_{B_0} = 1$	7
20)	$a_{B_0} = 0$	$a_{B_0} = 0$	-	$a_{B_0} = 0$	$a_{B_0} = 1$	7
21)	$a_{B_0} = 0$	$a_{B_0} = 0$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	8
22)	$a_{B_0} = 0$	$a_{B_0} = 0$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	8
23)	$a_{B_0} = 0$	$a_{B_0} = 0$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	8
24)	$a_{B_0} = 0$	$a_{B_0} = 0$	$a_{B_0} = 0$	$a_{B_0} = 1$	$a_{B_0} = 0$	8

In the following, it will be described how sequences can be encoded into two behaviour networks that are part of RAVON's control system (see [Armbrust 11a]). The first network executes turning manoeuvres (see Sec. 3.1.2.1), while the second one detects dead ends (see Sec. 3.1.2.2).

3.1.2.1 Turning Manoeuvre

The off-road robot RAVON is equipped with a 4WD (four-wheel drive) and two independently steerable axles. Each of the two has an Ackermann steering geometry. This allows for a high degree of manoeuvrability even on rough terrain. However, the robot is not able to turn on the spot. Hence, if the robot has to turn around in a narrow place between obstacles, a sequence of back and forth manoeuvres is necessary. During autonomous operation, these turning manoeuvres have to be executed by a subcomponent of RAVON's control system.

Following the behaviour-based approach, this subcomponent has been realised as a network of iB2C behaviours called *(G) Turn Around*. It is depicted in Fig. 3.8. The network consists of two parts (left and right) that both generate a sequence of alternating back and forth manoeuvres. One of the parts realises a turning to the left (i.e. the robot moves forward to the left and backward to the right), while the other realises a turning to the right (i.e. the robot moves forward to the right and backward to the left). For each of these two parts, the two behaviours *Orientation Activation (OA)* and *Orientation Deactivation (OD)* monitor the robot's angle to its target. Each of them gets active if the angle is above a certain threshold that is set as a parameter of the behaviour.

The activities of the four behaviours are calculated as follows:

$$a_{\text{OA (Left)}}(t) = \begin{cases} \iota_{\text{OA (Left)}}(t) & \text{if } \alpha_{\text{Target}}(t) \geq \alpha_{\text{Activate (Left)}} \\ 0 & \text{else} \end{cases} \quad (3.7)$$

$$a_{\text{OD (Left)}}(t) = \begin{cases} \iota_{\text{OD (Left)}}(t) & \text{if } \alpha_{\text{Target}}(t) \geq \alpha_{\text{Deactivate (Left)}} \\ 0 & \text{else} \end{cases} \quad (3.8)$$

$$a_{\text{OA (Right)}}(t) = \begin{cases} \iota_{\text{OA (Right)}}(t) & \text{if } \alpha_{\text{Target}}(t) \leq \alpha_{\text{Activate (Right)}} \\ 0 & \text{else} \end{cases} \quad (3.9)$$

$$a_{\text{OD (Right)}}(t) = \begin{cases} \iota_{\text{OD (Right)}}(t) & \text{if } \alpha_{\text{Target}}(t) \leq \alpha_{\text{Deactivate (Right)}} \\ 0 & \text{else} \end{cases} \quad (3.10)$$

$\alpha_{\text{Target}}(t)$ denotes RAVON's angle to the target at time t . The angle lies in $[-180^\circ, 180^\circ]$, where positive values correspond to the left side and negative ones to the right side.

$\alpha_{\text{Activate (Left)}}$, $\alpha_{\text{Deactivate (Left)}}$, $\alpha_{\text{Activate (Right)}}$, and $\alpha_{\text{Deactivate (Right)}}$ are parameters of *Orientation Activation (Left)*, *Orientation Deactivation (Left)*, *Orientation Activation (Right)*, and *Orientation Deactivation (Right)*, respectively. They have been set as follows:

$$\begin{aligned} \alpha_{\text{Activate (Left)}} &= 90^\circ & \alpha_{\text{Deactivate (Left)}} &= 45^\circ \\ \alpha_{\text{Activate (Right)}} &= -90^\circ & \alpha_{\text{Deactivate (Right)}} &= -45^\circ \end{aligned}$$

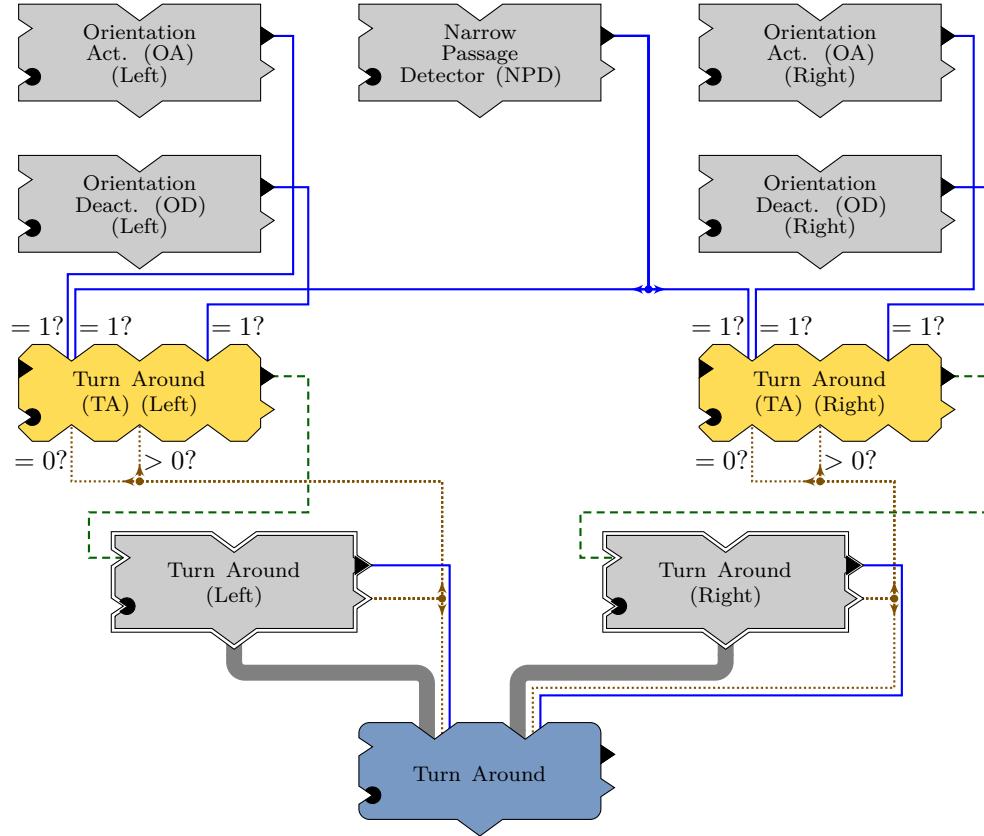


Figure 3.8: The behaviour-based network for turning around, (*G*) *Turn Around*. As usual, standard behaviours are represented by grey symbols, while CBS nodes are depicted by yellow ones, and the fusion behaviour is visualised by a blue symbol. Behaviour groups have double boundary lines. Behaviours with a stimulation port to which no other behaviour is connected are either permanently stimulated (black triangle) or stimulated from the outside.

Figure 3.8 shows that each of the two behaviours of one side is connected to a corresponding CBS. The activity of an instance of *Orientation Activation* is connected to a port associated with an enabling input condition, while the activity of an instance of *Orientation Deactivation* is connected to a port associated with a permanent input condition. In all cases, the CBS checks whether the input signal equals 1. The effect of this connection is a hysteresis: The CBS for the left side gets active if the robot's angle to the target rises above 90° and stays active until the angle falls below 45° . The CBS for the right side gets active accordingly. Assuming that *OA (Left)* and *OD (Left)* are fully stimulated and not inhibited (i.e. $\iota_{OA \text{ (Left)}} = \iota_{OD \text{ (Left)}} = 1$), the input relations for the left side are the following (cp. Eq. 3.1):

$$\left(\text{ir}_{(CBS) \text{ TA (Left)}}\right)_0(t) = \begin{cases} 1 & \text{if } a_{OA \text{ (Left)}}(t) = 1 \\ 0 & \text{else} \end{cases} = \begin{cases} 1 & \text{if } \alpha_{\text{Target}}(t) \geq \alpha_{\text{Activate (Left)}} \\ 0 & \text{else} \end{cases}$$

$$\left(\text{ir}_{(CBS) \text{ TA (Left)}}\right)_1(t) = \begin{cases} 1 & \text{if } a_{OD \text{ (Left)}}(t) = 1 \\ 0 & \text{else} \end{cases} = \begin{cases} 1 & \text{if } \alpha_{\text{Target}}(t) \geq \alpha_{\text{Deactivate (Left)}} \\ 0 & \text{else} \end{cases}$$

Another behaviour, the *Narrow Passage Detector (NPD)*, monitors the areas on the sides of the robot in order to determine whether the robot is situated in a narrow passage. If this is the case, the behaviour's activity rises to the value of its activation (usually 1) and stays at this level until the robot has moved to a less confined area again. The activity output port of the *Narrow Passage Detector* is connected to both CBS nodes and associated with enabling input conditions that check whether the activity equals 1. Only if this is the case can alternating back and forth turning manoeuvres be initiated. The reason for having this additional check is the execution of backward motions. As RAVON's main sensor systems (like the ones of many robots) are concentrated on its front, driving backwards shall be avoided if possible. In case the area around the robot is not occluded by obstacles, turning around while driving forwards can be expected to be possible, hence there is no need for backward motions. According to Eq. 3.1, the input relation corresponding to the *Narrow Passage Detector* can be written as follows:

$$\left(\text{ir}_{(CBS) \text{ TA (Left)}}\right)_2(t) = \begin{cases} 1 & \text{if } a_{NPD}(t) = 1 \\ 0 & \text{else} \end{cases}$$

With Eqs. 3.2 and 3.4, this yields for the input conditions of the CBS of the left side (*(CBS) Turn Around (Left)* or *(CBS) TA (Left)*):

$$\left(\text{ic}_{(CBS) \text{ TA (Left)}}\right)_1(t) = \begin{cases} 1 & \text{if } \alpha_{\text{Target}}(t) \geq \alpha_{\text{Deactivate (Left)}} \\ 0 & \text{else} \end{cases}$$

$$\begin{aligned}
& \left(\text{ic}_{(\text{CBS}) \text{ TA (Left)}} \right)_0(t) \\
&= \begin{cases} 1 & \text{if } \exists t_0 \leq t : \left(\left(\text{ir}_{(\text{CBS}) \text{ TA (Left)}} \right)_0(t_0) = 1 \right) \\ & \quad \wedge \left(\left(\text{ir}_{(\text{CBS}) \text{ TA (Left)}} \right)_2(t_0) = 1 \right) \\ & \quad \wedge \left(\left(\text{ic}_{(\text{CBS}) \text{ TA (Left)}} \right)_1(t_1) = 1 \quad \forall t_1 : t_0 \leq t_1 \leq t \right) \\ 0 & \text{else} \end{cases} \\
&= \begin{cases} 1 & \text{if } \exists t_0 \leq t : \left(\alpha_{\text{Target}}(t_0) \geq \alpha_{\text{Activate (Left)}} \right) \\ & \quad \wedge \left(a_{\text{NPD}}(t_0) = 1 \right) \\ & \quad \wedge \left(\alpha_{\text{Target}}(t_1) \geq \alpha_{\text{Deactivate (Left)}} \right) \\ & \quad \quad \forall t_1 : t_0 \leq t_1 \leq t \\ 0 & \text{else} \end{cases}
\end{aligned}$$

$$\begin{aligned}
& \left(\text{ic}_{(\text{CBS}) \text{ TA (Left)}} \right)_2(t) \\
&= \begin{cases} 1 & \text{if } \exists t_0 \leq t : \left(\left(\text{ir}_{(\text{CBS}) \text{ TA (Left)}} \right)_0(t_0) = 1 \right) \\ & \quad \wedge \left(\left(\text{ir}_{(\text{CBS}) \text{ TA (Left)}} \right)_2(t_0) = 1 \right) \\ & \quad \wedge \left(\left(\text{ic}_{(\text{CBS}) \text{ TA (Left)}} \right)_1(t_1) = 1 \quad \forall t_1 : t_0 \leq t_1 \leq t \right) \\ 0 & \text{else} \end{cases} \\
&= \begin{cases} 1 & \text{if } \exists t_0 \leq t : \left(\alpha_{\text{Target}}(t_0) \geq \alpha_{\text{Activate (Left)}} \right) \\ & \quad \wedge \left(a_{\text{NPD}}(t_0) = 1 \right) \\ & \quad \wedge \left(\alpha_{\text{Target}}(t_1) \geq \alpha_{\text{Deactivate (Left)}} \right) \\ & \quad \quad \forall t_1 : t_0 \leq t_1 \leq t \\ 0 & \text{else} \end{cases}
\end{aligned}$$

As $\left(\text{ic}_{(\text{CBS}) \text{ TA (Left)}} \right)_0(t) = \left(\text{ic}_{(\text{CBS}) \text{ TA (Left)}} \right)_2(t)$, the above equations can be summarised to the following (cp. Eq. 3.5):

$$\begin{aligned}
& a_{(CBS) TA (Left)}(t) = 1 \\
\Leftrightarrow & \iota_{(CBS) TA (Left)}(t) \cdot \left(\text{ic}_{(CBS) TA (Left)} \right)_0(t) \cdot \left(\text{ic}_{(CBS) TA (Left)} \right)_1(t) \cdot \left(\text{ic}_{(CBS) TA (Left)} \right)_2(t) \\
& = 1 \\
\Leftrightarrow & \left(\iota_{(CBS) TA (Left)}(t) = 1 \right) \wedge \left(\left(\text{ic}_{(CBS) TA (Left)} \right)_0(t) = 1 \right) \wedge \left(\left(\text{ic}_{(CBS) TA (Left)} \right)_1(t) = 1 \right) \wedge \\
& \left(\left(\text{ic}_{(CBS) TA (Left)} \right)_2(t) = 1 \right) \\
\Leftrightarrow & \left(\iota_{(CBS) TA (Left)}(t) = 1 \right) \wedge \left(\exists t_0 \leq t : \left((a_{NPD}(t_0) = 1) \wedge (\alpha_{Target}(t_0) \geq \alpha_{Activate (Left)}) \right. \right. \\
& \left. \left. (\alpha_{Target}(t_1) \geq \alpha_{Deactivate (Left)}) \forall t_1 : t_0 \leq t_1 \leq t \right) \right) \wedge \left(\alpha_{Target}(t) \geq \alpha_{Deactivate (Left)} \right) \\
\Leftrightarrow & \left(\iota_{(CBS) TA (Left)}(t) = 1 \right) \wedge \left(\exists t_0 \leq t : \left((a_{NPD}(t_0) = 1) \wedge (\alpha_{Target}(t_0) \geq \alpha_{Activate (Left)}) \right. \right. \\
& \left. \left. (\alpha_{Target}(t_1) \geq \alpha_{Deactivate (Left)}) \forall t_1 : t_0 \leq t_1 \leq t \right) \right)
\end{aligned}$$

Naturally, for a complete analysis of the activity of $(CBS) TA (Left)$, its feedback conditions would also have to be taken into account. If the input conditions of one of the two CBS nodes are fulfilled, the CBS in question will get active and stimulate a corresponding behaviour group ($(G) Turn Around (Left)$ or $(G) Turn Around (Right)$). These groups can execute turning manoeuvres by issuing appropriate motion commands. Each of them gets active if it is stimulated by the corresponding CBS and also raises its target rating, which is calculated based on the target ratings of two CBS nodes ($(CBS) Turn Around (Forward)$ and $(CBS) Turn Around (Backward)$) in the group (see Fig. 3.9). After the turning manoeuvre of a group has been completed, the group gets inactive and resets its target rating to 0. As can be seen in Fig. 3.8, the target rating is sent to the corresponding CBS and connected with two ports associated with an enabling and an ordering feedback condition, respectively. A check whether the target rating is > 0 belongs to the ordering feedback condition, while a check whether it is $= 0$ belongs to the enabling feedback condition. The equations for the feedback relations and conditions can be set up similarly to the above equations for the input relations and conditions. The result of these connections is that each CBS knows when its associated behaviour group has completed its turning manoeuvres. At this moment, the CBS will be reset to its initial state, stop stimulating the corresponding group, and restart checking its input conditions.

At the bottom of the behaviour network, a fusion behaviour realising a maximum fusion combines the motion commands that are output by the two behaviour groups. At one time, only one sequence of turning motions (to the left or to the right) is executed. Hence, the two groups cannot be active at the same time. This is guaranteed by the two CBS nodes and their input conditions. In case a group is active, its output values are forwarded by the fusion behaviour, while the outputs of the other (inactive) group are discarded.

The two behaviour groups $(G) Turn Around (Left)$ and $(G) Turn Around (Right)$ also make use of CBS nodes. Their network structure is depicted in Fig. 3.9 using the example of the group dealing with the left side. It consists of three CBS nodes ($(CBS) Turn Around (Forward)$, $(CBS) Turn Around (Backward)$, and $(CBS) Cycle Init$), three fusion behaviours

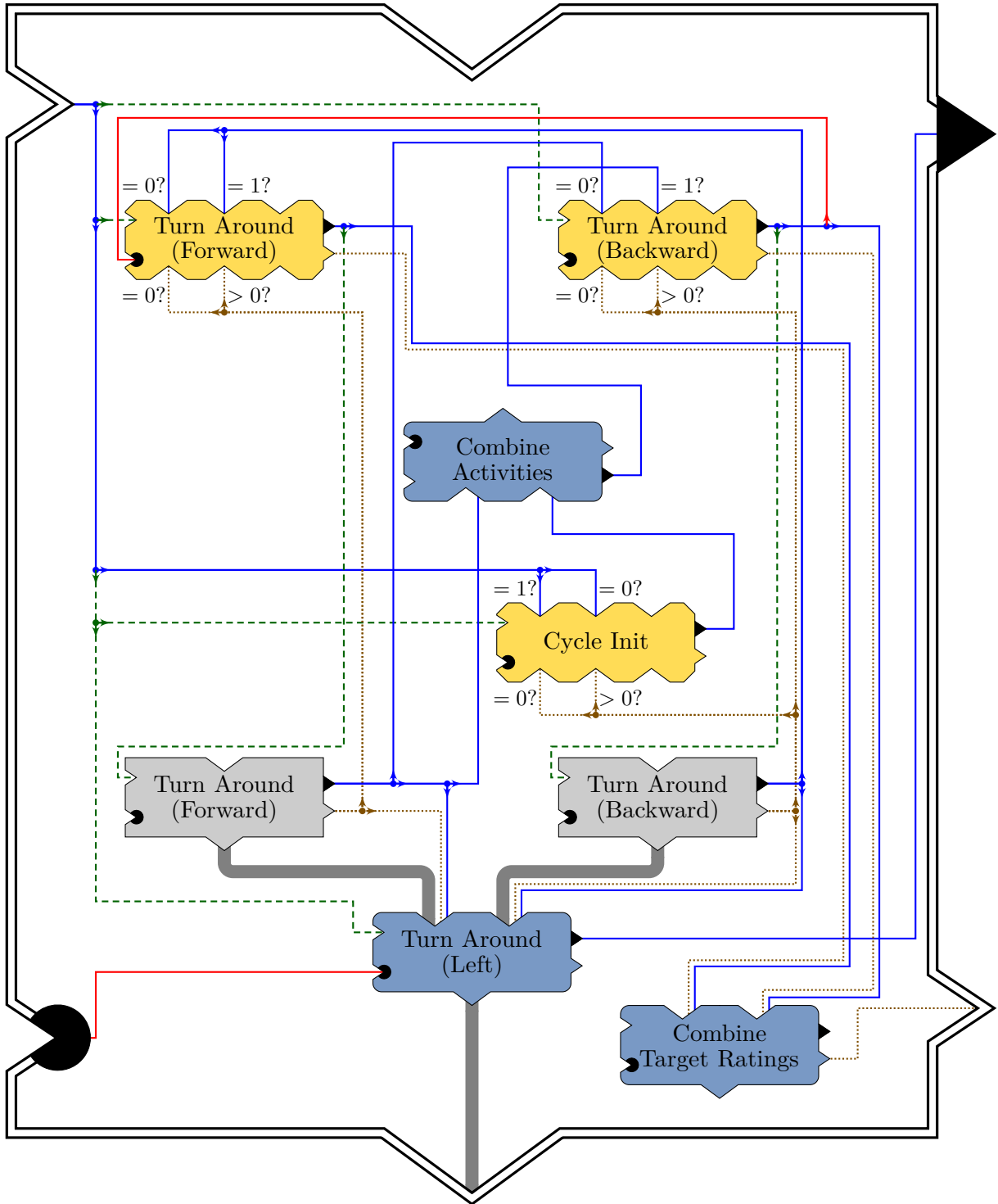


Figure 3.9: The behaviour-based network realising turning manoeuvres for the left side, (*G*) *Turn Around (Left)*. Another network (with the same structure) realises turning manoeuvres for the right side.

that realise maximum fusions (*(F) Turn Around (Left)*, *(F) Combine Activities*, and *(F) Combine Target Ratings*), as well as two standard behaviours (*Turn Around (Forward)* and *Turn Around (Backward)*). The two standard behaviours are responsible for calculating motion commands for the robot. *Turn Around (Forward)* outputs a desired *forward* velocity and a rotation value corresponding to the side (left or right) of the containing group. Accordingly, *Turn Around (Backward)* outputs a desired *backward* velocity and a rotation value. They both also perform additional checks like monitoring parts of the area around the robot to determine whether it is advisable to continue moving in one direction or to stop the robot's motion. Their outputs are combined using a maximum fusion and then sent to the output ports of the group. The remaining components of *Turn Around (Left)* and *(Right)* are responsible for switching between the two behaviours for forward and backward motion, respectively. The following enumeration shows a typical switching sequence:

1. *(CBS) Turn Around (Backward)* gets active and stimulates *Turn Around (Backward)*.
2. *Turn Around (Backward)*, unsatisfied with the situation, gets active and issues motion commands.
3. *Turn Around (Backward)* gets satisfied and inactive.
4. *(CBS) Turn Around (Backward)* gets inactive.
5. *(CBS) Turn Around (Forward)* gets active and stimulates *Turn Around (Forward)*.
6. *Turn Around (Forward)*, unsatisfied with the situation, gets active and issues motion commands.
7. *Turn Around (Forward)* gets satisfied and inactive.
8. *(CBS) Turn Around (Forward)* gets inactive.
9. *(CBS) Turn Around (Backward)* gets active and stimulates *Turn Around (Backward)*.
10. ...

The result is that RAVON alternately moves forwards and backwards while turning. This interaction between the involved behaviours originates from the connection of the activity and target rating output ports of *Turn Around (Forward)* and *Turn Around (Backward)* with input and feedback condition ports of *(CBS) Turn Around (Forward)* and *(CBS) Turn Around (Backward)*.

At the start of this cyclic sequence, however, neither of the two turning behaviours has been active yet. Hence, neither the ordering input condition of *(CBS) Turn Around (Forward)* nor the one of *(CBS) Turn Around (Backward)* is fulfilled. In order to initiate the cycle, another coordinating behaviour (*(CBS) Cycle Init*) has been added. It gets active when the stimulation of the group switches from 0 (ordering input condition) to 1 (enabling input condition). Its activity output is connected to the ordering input port of *(CBS) Turn Around (Backward)* via a fusion behaviour. As a result, the cycle can start with *(CBS) Turn Around (Backward)* getting active. As soon as *Turn Around (Backward)* has

been unsatisfied for the first time and gets satisfied again, *(CBS) Cycle Init* is reset. The activities and target ratings of *(CBS) Turn Around (Forward)* and *(CBS) Turn Around (Backward)* are combined using the third fusion behaviour. In case one of these two CBS nodes is active, its target rating is propagated through the fusion behaviour to the target rating output port of the group.

This small example can already be used to illustrate an advantage of encoding tasks (turning around) consisting of subtasks (turning while driving forwards, turning while driving backwards) into a network of behaviours: The activities of the involved behaviours directly provide information about the current state of the system. For example, in case the robot unexpectedly does not start executing back and forth manoeuvres or suddenly stops doing so, it is possible to find the cause of the maloperation by analysing the behaviours' activities. The frameworks MCA2-KL and FINROC provide special (tool) support for such analyses (see Chap. B), with which a developer or robot operator can easily get an overview of the fulfilment of the different CBS nodes and the activities and target ratings of the involved behaviours.

The following experiment shall illustrate the operation of the network for turning around using the simulation of RAVON, which is visualised in the MCAGUI (see Sec. B.1). Figure 3.10 depicts a sequence of manoeuvres executed by the simulated robot. The experiment starts with RAVON being placed inside a narrow corridor delimited by various obstacles like bushes and trees (see Fig. 3.10a). As there is not much space on the robot's sides, the *Narrow Passage Detector* assumes that the robot is in a narrow passage and gets active. At the beginning, RAVON is oriented towards the front opening of the corridor and is commanded to drive towards a target that is situated behind it on its left side outside of the passage. Hence, *Orientation Activation (Left)* and *Orientation Deactivation (Left)* are also active. The sequence shows how the *Turn Around* network turns the robot around with several back and forth movements. With the first movement, the robot drives back towards its right side until its back gets too close to the obstacles (3.10b). A forward movement towards the left follows (3.10c). It is stopped when RAVON's front gets too close to obstacles. During the forward motion, the robot's angle to the target α_{Target} falls below $\alpha_{\text{Activate (Left)}} = 90^\circ$. Hence, the activity of *Orientation Activation (Left)* $a_{\text{OA (Left)}}$ goes down to 0 according to Eq. 3.7. However, the activity of *Orientation Deactivation (Left)* $a_{\text{OD (Left)}}$ stays longer above 0 because α_{Target} stays longer above $\alpha_{\text{Deactivate (Left)}} = 45^\circ$. Due to this hysteresis, the turning manoeuvre continues (3.10d) until the robot faces the back opening of the passage (3.10e).

The sequential execution of forward and backward movements as described above can be considered as a sequential execution of tasks that is encoded in a behaviour network—with the particularity that the two tasks (forward movement and backward movement) are executed alternately. Naturally, this is a rather simple example of a task sequence. A more complex example is described in Sec. 3.2.2.

3.1.2.2 Dead End Detection

For an off-road robot navigating in unstructured environments, the ability to properly detect whether it has reached a dead end can significantly improve its navigation capabilities. In the control system of RAVON, a behaviour-based sub-network consisting of two CBS nodes (*(CBS) Robot in Passage* and *(CBS) Dead End Detected*) and three standard behaviours

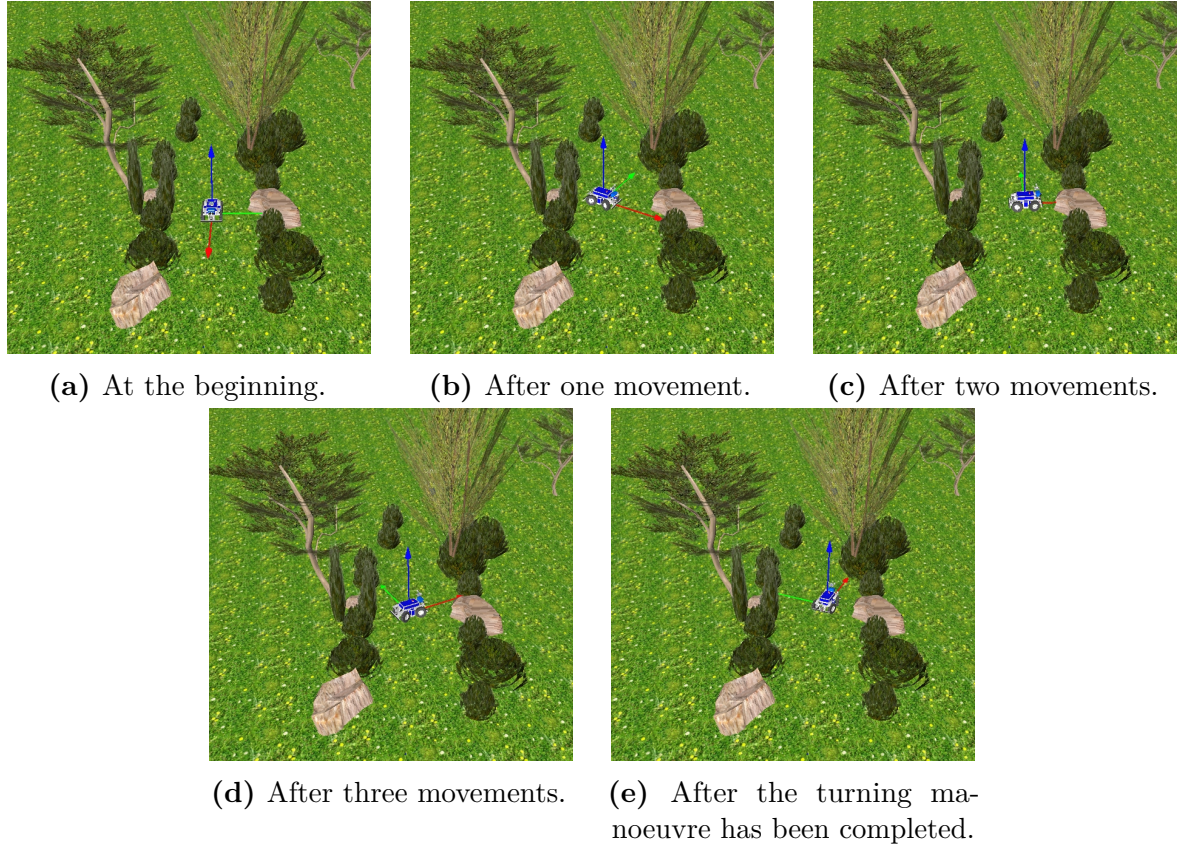


Figure 3.10: The behaviour-based network *Turn Around* makes RAVON turn around in a narrow passage. The red arrow points to RAVON's front, the green one to its left side, and the blue one upwards.

(*Narrow Passage Detector*, *Entering Passage*, and *Blockade Detector*) is responsible for this task. It is depicted in Fig. 3.11. In the following, it is explained how this network works.

An informal definition of a dead end is a place where the robot's way is blocked so that it cannot move on. In order to detect such a place automatically, a more formal definition of a dead end is needed. Therefore, in the context of this work, the robot is said to be in a dead end if it has driven into a structure consisting of the following three elements (see [Armbrust 11a]):

1. *Passage Entry*: an opening between two (formations of) obstacles at the beginning of the dead end
2. *Narrow Passage*: a corridor between obstacles in which the robot cannot drive left or right, but can only move on or back off
3. *Blockade*: an obstruction at the end of the passage that keeps the robot from moving on

In order to determine whether RAVON is situated in a dead end, the behaviour-based network has to be capable of detecting each single of the three elements forming a dead

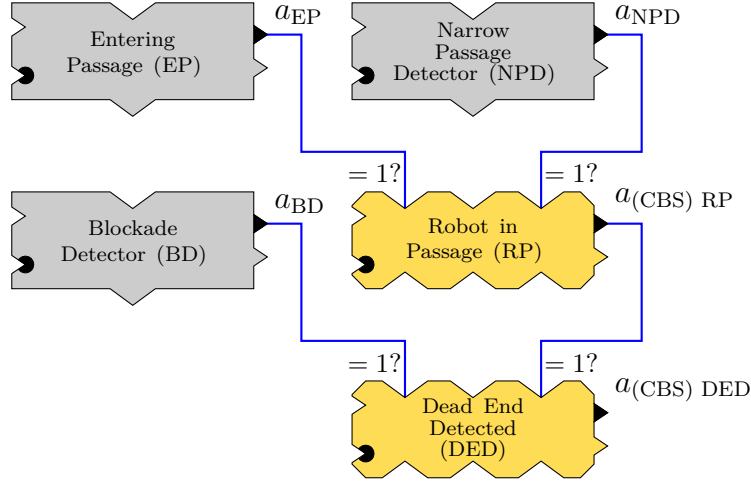


Figure 3.11: The behaviour-based network that recognises dead ends.

end. A combination of the detection of them is then interpreted as the detection of a dead end, i.e. the task of detecting a complex structure is divided into the combined tasks of detecting a number of simpler structures. The underlying idea is that in order to be situated in a dead end, the robot must have driven there first, which creates three conditions: The robot must have driven through a passage entry (first condition), it must have been within the corresponding narrow passage all the time since it entered the probable dead end (second condition), and the robot must be faced with a blockade of its path (third condition). The event of driving through a passage entry is especially notable because in order to find a way out of the dead end, it is necessary to know where it starts.

Each of the partial structures of a dead end shall be detected using a specific behaviour that gets active upon the presence of the corresponding partial structure (see Fig. 3.11). A high activity of *Entering Passage (EP)* shall indicate that the robot has driven through a passage's entry; if the *Narrow Passage Detector (NPD)* is active, the robot is within a narrow passage; finally, the presence of a blockade in front of the robot results in the *Blockade Detector (BD)* getting active. The activities of these three behaviours are defined as follows:

$$a_{EP}(t) = \iota_{EP}(t) \cdot \begin{cases} 1 & \text{if the robot has driven through a passage's entry before time } t \\ 0 & \text{else} \end{cases}$$

$$a_{NPD}(t) = \iota_{NPD}(t) \cdot \begin{cases} 1 & \text{if the robot is within a narrow passage at time } t \\ 0 & \text{else} \end{cases}$$

$$a_{BD}(t) = \iota_{BD}(t) \cdot \begin{cases} 1 & \text{if a blockade is detected in front of the robot at time } t \\ 0 & \text{else} \end{cases}$$

These three behaviours are combined with two CBS nodes (*(CBS) Robot in Passage* and *(CBS) Dead End Detected*) to a behaviour network. If RAVON has driven through a passage entry and is then situated within a narrow passage, the activity of *(CBS) Robot in Passage*

((CBS) RP) shall be 1. A high activity of ((CBS) Dead End Detected ((CBS) DED) shall indicate that the robot has driven into a dead end.

Entering Passage, *Narrow Passage Detector*, and *Blockade Detector* are connected to the CBS nodes according to the detection mechanism described above (see Fig. 3.11). The activity outputs of *Entering Passage* and *Narrow Passage Detector* are connected to an enabling and a permanent input port of ((CBS) Robot in Passage, respectively. According to Eqs. 3.2, 3.4, and 3.5, this yields for the activity of ((CBS) Robot in Passage:

$$\begin{aligned}
a_{(\text{CBS}) \text{ RP}}(t) &= 1 \\
&\Leftrightarrow \iota_{(\text{CBS}) \text{ RP}}(t) \cdot \text{ic}_{\text{EP}}(t) \cdot \text{ic}_{\text{NPD}}(t) = 1 \\
&\Leftrightarrow \left(\iota_{(\text{CBS}) \text{ RP}}(t) = 1 \right) \wedge \left(\text{ic}_{\text{EP}}(t) = 1 \right) \wedge \left(\text{ic}_{\text{NPD}}(t) = 1 \right) \\
&\Leftrightarrow \left(\iota_{(\text{CBS}) \text{ RP}}(t) = 1 \right) \wedge \left(\exists t_0 \leq t : ((a_{\text{EP}}(t_0) = 1) \wedge (a_{\text{NPD}}(t_1) = 1 \ \forall t_1 : t_0 \leq t_1 \leq t)) \right) \wedge \\
&\quad \left(a_{\text{NPD}}(t) = 1 \right) \\
&\Leftrightarrow \left(\iota_{(\text{CBS}) \text{ RP}}(t) = 1 \right) \wedge \left(\exists t_0 \leq t : ((a_{\text{EP}}(t_0) = 1) \wedge (a_{\text{NPD}}(t_1) = 1 \ \forall t_1 : t_0 \leq t_1 \leq t)) \right)
\end{aligned} \tag{3.11}$$

The activity outputs of ((CBS) Robot in Passage and *Blockade Detector* are connected to an enabling and a permanent input port of ((CBS) Dead End Detected, respectively. Again according to Eqs. 3.2, 3.4, and 3.5, this yields for the activity of ((CBS) Dead End Detected:

$$\begin{aligned}
a_{(\text{CBS}) \text{ DED}}(t) &= 1 \\
&\Leftrightarrow \iota_{(\text{CBS}) \text{ DED}}(t) \cdot \text{ic}_{\text{BD}}(t) \cdot \text{ic}_{\text{RP}}(t) = 1 \\
&\Leftrightarrow \left(\iota_{(\text{CBS}) \text{ DED}}(t) = 1 \right) \wedge \left(\text{ic}_{\text{BD}}(t) = 1 \right) \wedge \left(\text{ic}_{\text{RP}}(t) = 1 \right) \\
&\Leftrightarrow \left(\iota_{(\text{CBS}) \text{ DED}}(t) = 1 \right) \wedge \left(\exists t_0 \leq t : ((a_{\text{BD}}(t_0) = 1) \wedge \right. \\
&\quad \left. \left(a_{(\text{CBS}) \text{ RP}}(t_1) = 1 \ \forall t_1 : t_0 \leq t_1 \leq t \right) \right) \wedge \left(a_{(\text{CBS}) \text{ RP}}(t) = 1 \right) \\
&\Leftrightarrow \left(\iota_{(\text{CBS}) \text{ DED}}(t) = 1 \right) \wedge \left(\exists t_0 \leq t : ((a_{\text{BD}}(t_0) = 1) \wedge \right. \\
&\quad \left. \left(a_{(\text{CBS}) \text{ RP}}(t_1) = 1 \ \forall t_1 : t_0 \leq t_1 \leq t \right) \right)
\end{aligned} \tag{3.12}$$

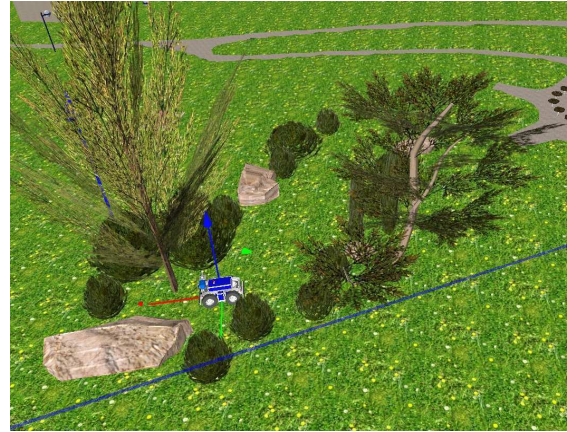
Using Eq. 3.11 to replace $a_{(\text{CBS}) \text{ RP}}(t_1) = 1$ in Eq. 3.12 yields:

$$\begin{aligned}
a_{(\text{CBS}) \text{ DED}}(t) &= 1 \\
&\Leftrightarrow \left(\iota_{(\text{CBS}) \text{ DED}}(t) = 1 \right) \wedge \left(\exists t_0 \leq t : \left((a_{\text{BD}}(t_0) = 1) \wedge \left(\left(\iota_{(\text{CBS}) \text{ RP}}(t_1) = 1 \right) \wedge \right. \right. \right. \\
&\quad \left. \left. \left(\exists t_2 \leq t_1 : ((a_{\text{EP}}(t_2) = 1) \wedge (a_{\text{NPD}}(t_3) = 1 \ \forall t_3 : t_2 \leq t_3 \leq t_1)) \right) \ \forall t_1 : t_0 \leq t_1 \leq t \right) \right)
\end{aligned} \tag{3.13}$$

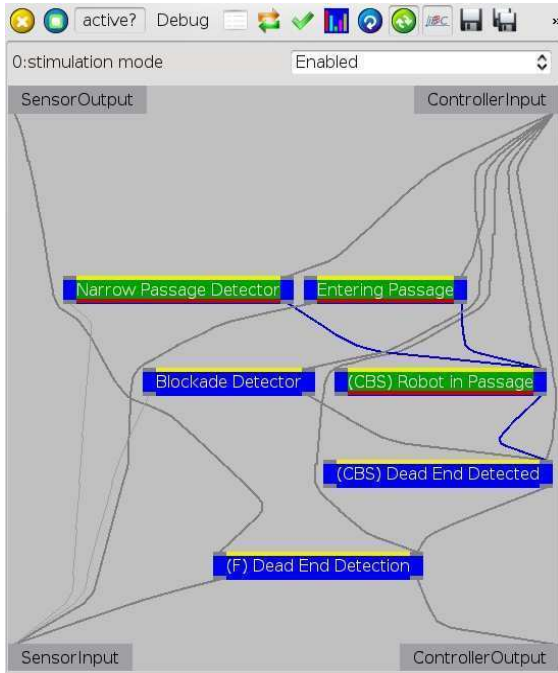
In the following, the operation of the detection network in the simulation of the off-road vehicle RAVON is presented (see [Armbrust 11a]).



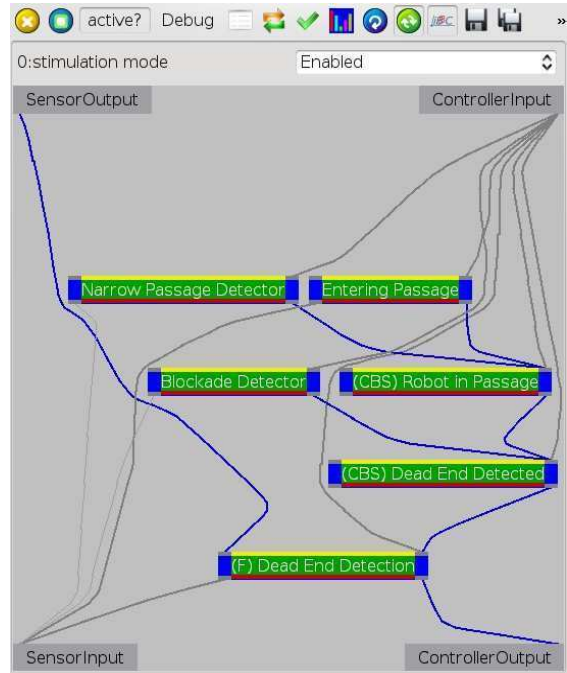
(a) RAVON has just entered the narrow passage.



(b) RAVON has reached the blockade at the end of the passage.



(c) The behaviours *Entering Passage* and *Narrow Passage* are fully active. Hence, the CBS *Robot in Passage* also is. The fusion behaviour (marked with “(F)”) determines this network’s activity and target rating.



(d) The behaviour *Blockade Detector* is now also active, resulting in a high activity of the CBS *Dead End Detected*. The group’s fusion behaviour is also active, signalling external components that the robot has driven into a dead end.

Figure 3.12: The behaviour-based network *Dead End Detection* recognises that RAVON has driven into a dead end. Figures a and b depict two situations that occurred while the robot was driving into the dead end. The state of the network in these two situations is illustrated by Figs. c and d. Each blue rectangle symbolises a behaviour. The yellow, green, and red bars visualise a behaviour’s activation, activity, and target rating, respectively (source: [Armbrust 11a]).

Figures 3.12a and 3.12b depict RAVON in a simulated off-road environment, visualised in the MCAGUI (see Sec. B.1). It contains a similar obstacle formation as the one described in Sec. 3.1.2.1. In Fig. 3.12a, the robot has just entered a narrow passage that is formed by a number of trees, bushes, and rocks. The end of the passage is blocked by a large rock, making it a dead end. At RAVON's current position, the rock is beyond its sensor range, i.e. it cannot detect it yet. The robot is given the command to drive towards a target that is situated ahead of it, but outside of the passage. Hence, the robot moves further ahead, driving deeper into the passage until it detects the blockade (see Figure 3.12b).

Figures 3.12c and 3.12d depict visualisations of the behaviour network in the two situations. The visualisations are generated by the MCABrowser (see Sec. B.1). Each blue rectangle symbolises a behaviour. The yellow, green, and red bars visualise a behaviour's activation, activity, and target rating, respectively. If a bar is not visible, the corresponding value is 0. At full length, a bar represents a value of 1. As can be seen in Fig. 3.12c, the behaviours *Entering Passage* and *Narrow Passage Detector* are fully active when the robot has just entered the passage. The figure also shows that the activation of *(CBS) Robot in Passage* is 1. Hence, *(CBS) Robot in Passage* is also active in compliance with Eq. 3.11. This indicates that RAVON is situated in a passage belonging to the passage entry through which it has just driven. Figure 3.12d depicts the behaviour network after the robot has driven deeper into the narrow passage and the blockade has come into its sensor range (cp. Fig. 3.12b). The activity of the *Blockade Detector* has risen to 1, resulting in *(CBS) Dead End Detected* also being active. This conforms to Eqs. 3.12 and 3.13.

The two above examples of the application of the CBS node originate from the control system of the autonomous off-road robot RAVON. How the CBS can be used in the control system of the autonomous bucket excavator THOR will be described in Chap. 5. In [Hirth 12], the coordination of behaviours in the control system of the humanoid robot ROMAN using CBS nodes is presented.

3.1.3 Discussion

According to Design Decision 2, the developed concepts for realising sequences shall be integrated into the behaviour architecture iB2C. This section has started with this task by showing how behaviour activity sequences can be realised in iB2C behaviour networks. For this purpose, a novel coordination behaviour (the CBS node) has been introduced as an extension of the behaviour-based architecture iB2C. This extension allows for establishing special inter-behaviour connections as mentioned in Design Decision 3. The two examples presented in this section have shown that comparably simple behaviour networks containing a number of CBS nodes can realise the sequential execution of actions as well as detect the sequential perception of structures in the environment. With this technique, a basis for the encoding of task sequences within a behaviour network in a decentralised way (cp. Design Decision 3) has been created.

However, the two simple examples already show some limitations of the approach presented so far. The first limitation is that the sequences had not been formally defined before the creation of the networks. This contradicts Design Decision 1, according to which sequences shall be represented using Moore machines. In the case of more complex sequences, not formally defining the sequence to be realised can easily be a source of errors.

The second limitation is that the design of the networks did not follow strict guidelines. Due to the various options of connecting behaviours in general and the complexity of the CBS node in particular, it would have been possible to realise the same task with a different network. Designing a control system in such a way is highly dependant on the experience of the developer. This is especially a challenge for novices who lack long-term experience with behaviour-based systems. In such a case, the development process is especially tedious and error-prone. While this problem may be manageable for smaller systems, it can represent a major hurdle when implementing more complex tasks.

In order to address these limitations, Sec. 3.2 will present a concept for the structured design of behaviour-based systems realising complex tasks. This concept will be based on the definition of a task sequence as a Moore machine, which will be transferred into a corresponding iB2C network that encodes the task sequence defined by the Moore machine (cp. Design Decision 3). In accordance with Design Decision 4, graphical tool support will be provided.

3.2 Transferring Moore Automata into iB2C Networks

In the previous section, it has been explained how behaviour activity sequences can be realised in iB2C networks. It has also been explained that sound guidelines on how to structure such networks are needed. Otherwise, the quality of the network will highly depend on the experience of the developer and on his personal preference. This section describes a formal method for deriving the structure of an iB2C network from a Moore machine that defines a complex task consisting of sequences of subtasks.

3.2.1 Transformation Process

In order to facilitate the design of a behaviour-based system that shall realise a complex task, a highly structured description of the task is needed. An (ideally automatic) process shall transform this description into the structure of a corresponding behaviour network. Under these premises, one of the first questions that arises is in which form the description shall be provided. Finite-state machines (FSMs) have already been mentioned in Sec. 2.1.1 as having a number of advantages over other representations of complex tasks consisting of a number of subtasks: For developers with a computer science background, they are a usual representation of sequences. However, people who shall use the developed system typically lack a computer science background. But the graphic representation of FSMs makes them comparably easy to understand for these people. Moreover, FSMs can be easily processed by algorithms (and thus transformed into, e.g., a behaviour network) as their structure can be strictly defined.

Figure 3.13 depicts the proposed workflow. Its central element is the automatic transformation of an FSM that defines a complex task into the skeleton of a corresponding iB2C network. The first step of the workflow is to define the complex task in question manually as an FSM. Two groups of people can be involved in this task: The role of the *end users* is to define which task the system shall realise. They have extensive knowledge about the application domain, but may lack the ability to precisely express their requirements as an FSM. Hence, they are assisted by the *main developer*, who is responsible for the

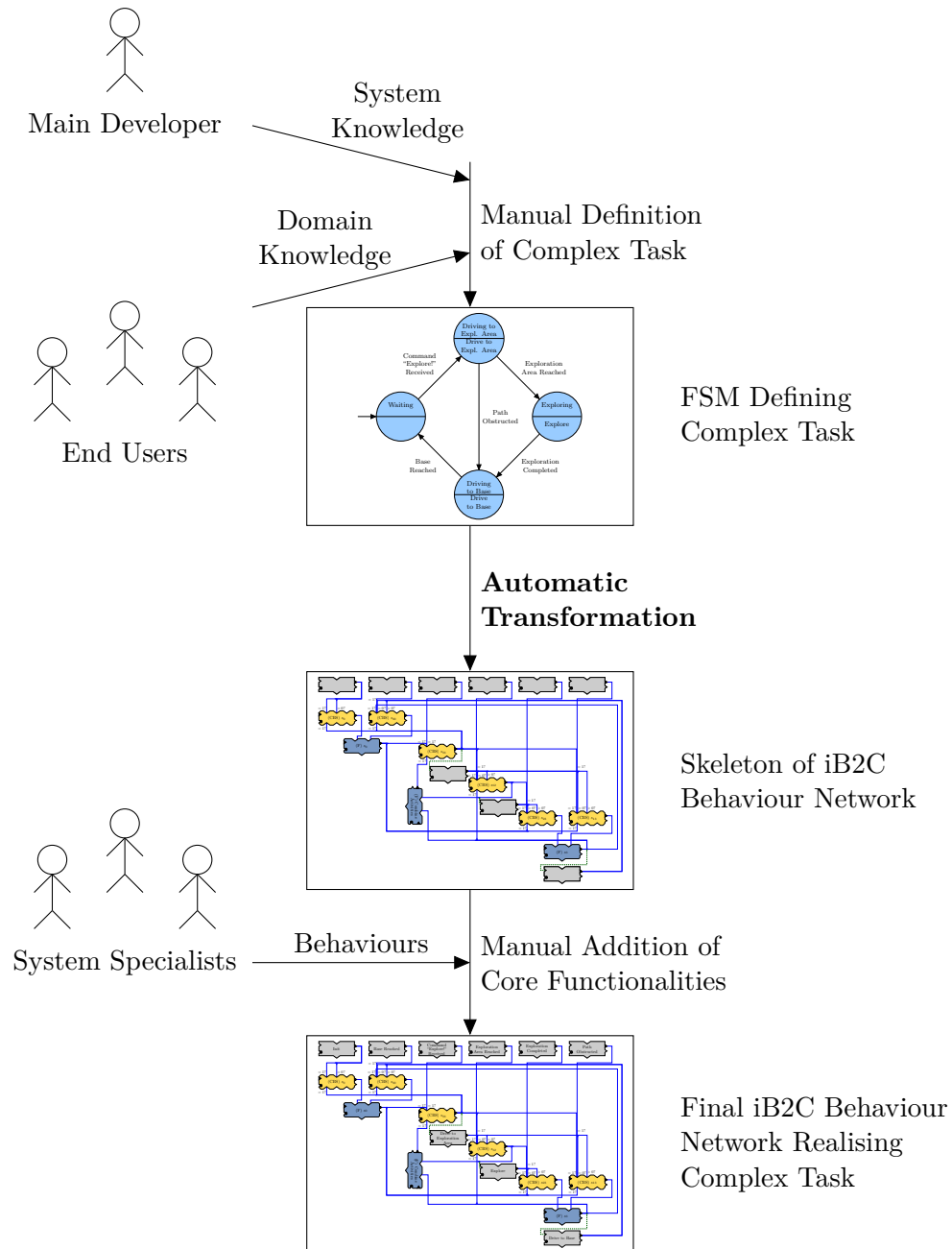


Figure 3.13: The proposed concept for the design of an iB2C behaviour network that fulfils a complex task defined by an FSM. Its central element is the automatic transformation of an FSM defining a complex task into the skeleton of a corresponding iB2C network.

complete system and provides his or her knowledge (e.g. about aspects of already existing software components or the specification of the available hardware). It is possible that the end users possess enough knowledge about the robotic system that they can create the FSM without the help of a developer. Furthermore, the developer could also be the person that will use the system. In this case, there would not be a special group of end users. The result of this first development step is an FSM that exactly defines the sequence of subtasks (or sequences of subtasks) which the complex task is composed of.

In the next step, the previously created FSM is automatically transformed into the skeleton of an iB2C behaviour network. This is the essential step of the proposed workflow. The term skeleton is used here as the functionality of the behaviours in the network has not been implemented yet—with the exception of the CBS nodes and the fusion behaviours needed for realising the sequential activation of the behaviours. The connections between the ports related to behaviour signals have been drawn at this stage. This is also necessary for the sequential execution to work. Having a fully automatic transformation process is possible because of a strict definition of the FSM representing a task and the availability of an algorithm transforming it into the skeleton of a corresponding iB2C behaviour network. Such an algorithm is described further below (see Algs. 3.1 and 3.2).

The third step, finally, consists of the manual addition of the core functionalities of the behaviours. This is done by so-called *system specialists* and can be supervised by the main developer. These are people that have very detailed knowledge about the robotic system or at least about parts of it. Their task is to implement specific sub-functionalities. Due to the distributed nature of behaviour-based systems, it is easy to parallelise the implementation work. Another advantage of this approach is that system specialists with different areas of expertise (e.g. low-level control, mapping, or path planning) can work jointly on the implementation. With the application of the behaviour-based approach, there is no need for having an expert for all aspects of the system. Different behaviours realise different sub-functionalities. Hence, different skills are needed for the implementation. As the structure of the iB2C behaviour network has already been created in the previous step, no system specialist with specific knowledge about the realisation of behaviour-based systems is needed. The result of the final step is an iB2C behaviour network realising the complex task defined by an FSM.

An algorithm executing the second of the above-mentioned steps is described in the following. A previous version has been published in [Armbrust 12b]. In Sec. 2.1.1, it has already been explained that there are differing definitions of the term (finite-)state machine. According to Design Decision 1, sequences shall be represented using Moore machines. Hence, in the remainder of this section, the term FSM shall always refer to a Moore machine as defined in Def. 2.5 unless otherwise indicated, i.e. an FSM is a 6-tuple $(S, s_I, \Sigma, \Lambda, T, G)$, with S being a set of states, $s_I \in S$ an initial state, Σ an input alphabet, Λ an output alphabet, $T : S \times \Sigma \rightarrow S$ a transition function, and $G : S \rightarrow \Lambda$ an output function. As described in Sec. 2.1.1, S corresponds to the set of subtasks the robot shall perform, Σ to certain conditions that have to be fulfilled before the robot can work on a subtask, and Λ to the set of actions that the robot can execute while working on the subtasks. Furthermore, T defines when the robot can stop working on one subtask and start working on another one, while G defines which action is executed while working on a particular subtask. Algorithms 3.1 and 3.2 describe how an iB2C behaviour network can be created that realises the sequential execution of subtasks defined by an FSM.

Algorithm 3.1: Transferring an FSM into an iB2C behaviour network (part 1).

```

input : a Moore machine  $M = (S, s_I, \Sigma, \Lambda, T, G)$ 
output: an iB2C network representing  $M$ 

1 foreach  $s_i \in S$  do                                     // iterate over all states
2   if  $G(s_i) = \lambda \neq \varepsilon$  then                       // Is there a subtask associated with this state?
3      $sb_i = \text{CreateBehaviour}(\lambda)$ ;
4   end
5   foreach  $s_j \in S, \sigma \in \Sigma$  with  $T(s_j, \sigma) = s_i$  do // iterate over all predecessor states
6      $cbs_{ji} = \text{CreateCBS}(\text{"(CBS) } s_{ji}\text{"})$ ;
7      $C_i = C_i \cup cbs_{ji}$ ;
8   end
9   if  $s_i == s_I$  then                                     // Current state is initial state?
10     $ib_i = \text{CreateBehaviour}(\text{"Init"})$ ;
11     $cbs_i = \text{CreateCBS}(\text{"(CBS) } s_i\text{"})$ ;
12     $C_i = C_i \cup cbs_{ji}$ ;
13     $\text{AddCondition}(ib_i, cbs_i, \text{"Enabling Input"}, \text{"= 1?"})$ ;
14     $\text{AddCondition}(ib_i, cbs_i, \text{"Ordering Input"}, \text{"= 0?"})$ ;
15  end
16  if  $|C_i| == 0$  then                                     // No predecessor and not initial state?
17     $cbs_i = \text{CreateCBS}(\text{"(CBS) } s_i\text{"})$ ;
18     $C_i = C_i \cup cbs_i$ ;
19  end
20  if  $|C_i| == 1$  with  $C_i == \{cbs\}$  then // Is there only one transition leading to this
    state?
21    if  $G(s_i) \neq \varepsilon$  then                           // Subtask?
22       $\text{AddStimulation}(cbs, sb_i)$ ;
23    end
24  end
25  else                                                     // more than one transition leading to this state
26     $fb_i = \text{CreateFusionBehaviour}(\text{"(F) } s_i\text{"})$ ;
27    foreach  $cbs \in C_i$  do
28       $\text{AddActivityConnection}(cbs, fb_i)$ ;
29    end
30    if  $G(s_i) \neq \varepsilon$  then                           // Subtask?
31       $\text{AddStimulation}(fb_i, sb_i)$ ;
32    end
33  end
34  foreach  $s_j \in S, \sigma \in \Sigma$  with  $T(s_j, \sigma) = s_i$  do // iterate over conditions of all
    transitions leading to this state
35     $ib_{ji} = \text{CreateBehaviour}(\sigma)$ ;
36     $\text{AddCondition}(ib_{ji}, cbs_{ji}, \text{"Enabling Input"}, \text{"= 1?"})$ ;
37  end
38 end

```

Algorithm 3.2: Transferring an FSM into an iB2C behaviour network (part 2).

```

39 foreach  $s_i \in S$  do // iterate over all states
40   foreach  $s_j \in S, \sigma \in \Sigma$  with  $T(s_j, \sigma) = s_i$  do // iterate over all predecessor states
41     if  $|C_j| == 1$  with  $C_i == \{cbs\}$  then // Only one transition leading to predecessor?
42       AddCondition(cbs, cbsji, "Enabling Input", "= 1?");
43     end
44     else // more than one transition leading to predecessor
45       AddCondition(fbj, cbsji, "Enabling Input", "= 1?");
46     end
47     if  $G(s_j) \neq \varepsilon$  then // Subtask?
48       AddCondition(sbj, cbsji, "Enabling Input", "= 0?");
49       AddCondition(sbj, cbsji, "Ordering Input", "> 0?");
50     end
51     foreach cbs  $\in C_j$  do // iterate over all CBS nodes associated with predecessor states
52       if  $|\{s \in S : T(s_j, \sigma) = s, \sigma \in \Sigma\}| \leq 1$  then
53         if  $|C_i| == 1$  then
54           AddCondition(cbsji, cbs, "Enabling Feedback", "= 1?");
55         end
56         else
57           AddCondition(fbi, cbs, "Enabling Feedback", "= 1?");
58         end
59       end
60     end
61   end
62 end

63 foreach  $s_i \in S$  do // iterate over all states
64   if  $|\{s \in S : T(s_i, \sigma) = s, \sigma \in \Sigma\}| > 1$  then
65     fbfci = CreateFusionBehaviour("(F) Combine Activities  $s_i$ ");
66     foreach  $s_j \in S, \sigma \in \Sigma$  with  $T(s_i, \sigma) = s_j$  do // iterate over all successor states
67       if  $|C_j| == 1$  then // Only one transition leading to successor?
68         AddActivityConnection(cbsij, fbfci);
69       end
70       else // more than one transition leading to successor
71         AddActivityConnection(fbj, fbfci);
72       end
73     end
74     foreach  $s_j \in S, \sigma \in \Sigma$  with  $T(s_j, \sigma) = s_i$  do // iterate over all predecessor states
75       AddCondition(fbfci, cbsji, "Enabling Feedback", "= 1?");
76     end
77     if  $s_i == s_I$  then // Current state is initial state?
78       AddCondition(fbfci, cbsi, "Enabling Feedback", "= 1?");
79     end
80   end
81 end

```

The following list presents the basic ideas of Algs. 3.1 and 3.2 (cp. [Armbrust 12b]).

- For each state $s_i \in S$ of the FSM and each transition leading from a state s_j to s_i , a corresponding CBS node cbs_{ji} is created (see Line 6). All CBS nodes belonging to s_i form the set C_i . In case only one transition leads to s_i , cbs_{ji} being active is interpreted as the FSM being in state s_i , i.e. as the robot working on the subtask corresponding to s_i . If there are several transitions leading to a state s_i , this state is represented by a fusion behaviour fb_i (see Line 26) that combines all CBS nodes in C_i (see Line 28).
- For each condition $\sigma \in \Sigma$ on a transition from state s_j to a state s_i , a corresponding input behaviour ib_{ji} is created (see Line 35). This behaviour is active if and only if the condition is fulfilled. It is possible that a transition does not possess an explicit condition, but only the implicit condition that the subtask corresponding to the outgoing state has been fulfilled before the transition is executed. In this case, no input behaviour is created.
- For each subtask $\lambda \in \Lambda$ belonging to a state s_i , a corresponding stimulated behaviour sb_i is created (see Line 3). This behaviour executes the actions corresponding to this task and is stimulated by the CBS (see Line 22) or the fusion behaviour (see Line 31) representing the task. sb_i stays active as long as it works on executing its actions and is not inhibited by another behaviour. There may be subtasks for which the robot does not have to execute any action, i.e. no output is defined in the corresponding state ($\lambda = \varepsilon$). In this case, there is no behaviour sb_i .
- A transition from a state s_j with corresponding CBS cbs to a state s_i is represented by cbs getting inactive and cbs_{ji} becoming active. cbs_{ji} gets active if and only if:
 1. cbs is active (see Line 42).
 2. sb_j has been active (see Line 49) and is now inactive (i.e. has executed its actions) (see Line 48).
 3. All behaviours encapsulating relevant conditions are active (see Line 36).

The corresponding checks are performed using input conditions of cbs_{ji} . When cbs_{ji} gets active, it will signal cbs via a feedback condition of the latter to get inactive (see Line 54).

- If the start state s_j of a transition to s_i is represented by several CBS nodes and a fusion behaviour fb_j , the activity of fb_j being 1 is used as an enabling input condition for cbs_{ji} (see Line 45).
- If the target state s_i of a transition from a state s_j represented by the CBS cbs is represented by several CBS nodes and a fusion behaviour fb_i , the activity of fb_i being 1 is used as an enabling feedback condition for cbs (see Line 57).
- If a state s_i has more than one successor state, the activities of all nodes representing the successor states s_j have to be combined by a fusion behaviour fbfc_i (see Lines 68 and 71) before they can be used in an enabling feedback condition of the CBS node(s) (see Lines 75 and 78) representing s_i .

- For the initial state, a special input behaviour realising the initiation of the task sequence is created (see Line 10) and connected to the CBS representing the initial state (see Lines 13 and 14).

The algorithm uses several calls to methods in order to perform its task. Their functionality shall be explained in the following. By calling one of the methods `CreateBehaviour(<Name>)`, `CreateFusionBehaviour(<Name>)`, and `CreateCBS(<Name>)`, a behaviour of the respective type is created. A call of `AddStimulation(<Source>, <Target>)` connects the activity output port of `<Source>` with the stimulation input port of `<Target>`, resulting in a stimulating connection from behaviour `<Source>` to behaviour `<Target>`. The connection of a behaviour `<Source>` to a fusion behaviour `<Target>` is realised by calling `AddActivityConnection(<Source>, <Target>)`. As a result, `<Target>` will combine the activities of all behaviours connected in this way. Finally, calling `AddCondition(<Source>, <Target>, <Type>, <Relation>)` creates a new condition at a CBS node `<Target>` and connects the corresponding port with the activity output port of a given behaviour `<Source>`. The type of the condition (enabling, ordering, or permanent input or feedback) is given by the argument `<Type>`, while the argument `<Relation>` determines to which value and how the activity value of `<Source>` is compared.

Naturally, the specification of a high-level task using an FSM is common in robotics and thus described in several places in the literature. The authors of [Loetzsch 06] and [Risler 08], for example, describe how complex robot behaviour can be specified with hierarchical FSMs using their specification language XABSL³. Depending on the current state of an FSM, so-called *basic behaviours* can be activated. Several of the ideas presented in this section resemble aspects of the approach described in [Loetzsch 06] and [Risler 08]. A major difference is that in XABSL, there is a distinction between finite-state machines and basic behaviours, while in the work described in this thesis, an FSM is realised as a network of behaviours. Furthermore, the iB2C features the advantage that all behaviours share a common interface, which facilitates analysis and verification.

In the robot control architecture Saphira (see [Konolige 97b]), the language Colbert (see [Konolige 97a]) is used to define the behaviour of the sequencing layer, which in Saphira deals with the initiation and monitoring of behaviours as well as with taking care of temporal aspects of behaviour coordination. Colbert is based on FSAs, which are defined via procedure definitions using a subset of the ANSI C language along with a number of extensions for the control of robots. Again, FSMs are used to sequence behaviours, but are not implemented as behaviour networks.

In summary, the advantage of the approach presented here is that the developer can specify a sequence of tasks in the common way of using FSMs and—after the creation of a corresponding behaviour network—benefit from the advantages of the behaviour-based approach.

3.2.2 Example Application: Exploration Task

The presented concept for building an iB2C behaviour network realising a task that has been specified using an FSM shall be illustrated in the following with an extended version of the exploration task already presented in Sec. 2 as an example.

³XABSL: Extensible Agent Behavior Specification Language

Again, an autonomous mobile robot shall navigate to an unexplored area, execute an exploration task there, and then return to its base. This means that the robot has to execute the following three subtasks for fulfilling the exploration task:

$$\begin{aligned} T_0 &= \text{Drive to Exploration Area} \\ T_1 &= \text{Explore} \\ T_2 &= \text{Drive to Base} \end{aligned}$$

In contrast to the exploration task described in Sec. 2, it is possible that the path to the unexplored area is blocked this time. In this case, the robot will not perform the exploration, but directly drive back to its base. Hence, there are two possible task sequences—one in which the robot succeeds (s_{Success}) and one in which it fails to explore the previously unexplored area (s_{Failure}). According to Def. 2.3, they can be written as follows:

$$\begin{aligned} s_{\text{Success}0} &= T_0 = \text{Drive to Exploration Area} \\ s_{\text{Success}1} &= T_1 = \text{Explore} \\ s_{\text{Success}2} &= T_2 = \text{Drive to Base} \end{aligned}$$

$$\begin{aligned} s_{\text{Failure}0} &= T_0 = \text{Drive to Exploration Area} \\ s_{\text{Failure}1} &= T_1 = \text{Drive to Base} \end{aligned}$$

A Moore machine $(S, s_I, \Sigma, \Lambda, T, G)$ that represents these two sequences can be defined as follows:

$$S = \{s_0, s_1, s_2, s_3\} \text{ with } s_0 = \text{Waiting}, s_1 = \text{Driving to Exploration Area}, s_2 = \text{Exploring}, \\ s_3 = \text{Driving to Base}$$

$$s_I = s_0 = \text{Waiting}$$

$$\Sigma = \{\text{Command "Explore!" Received, Path Obstructed, Exploration Area Reached,} \\ \text{Exploration Completed, Base Reached}\}$$

$$\Lambda = \{\text{Drive to Exploration Area, Explore, Drive to Base}\}$$

$$\begin{aligned} T : T(\text{Waiting, Command "Explore!" Received}) &= \text{Driving to} \\ &\quad \text{Exploration Area} \\ T(\text{Driving to Exploration Area, Exploration Area Reached}) &= \text{Exploring} \\ T(\text{Driving to Exploration Area, Path Obstructed}) &= \text{Driving to Base} \\ T(\text{Exploring, Exploration Completed}) &= \text{Driving to Base} \\ T(\text{Driving to Base, Base Reached}) &= \text{Waiting} \end{aligned}$$

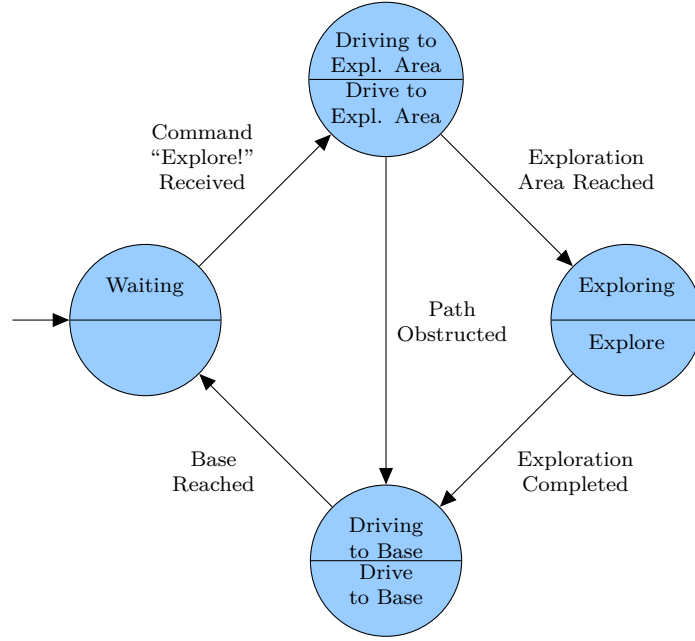


Figure 3.14: A state diagram visualising a Moore machine that realises the extended example of a robot exploring an area. The initial state, “Waiting”, is marked with an arrow without origin.

$$\begin{aligned}
 G : G(\text{Waiting}) &= \varepsilon \\
 G(\text{Driving to Exploration Area}) &= \text{Drive to Exploration Area} \\
 G(\text{Exploring}) &= \text{Explore} \\
 G(\text{Driving to Base}) &= \text{Drive to Base}
 \end{aligned}$$

Figure 3.14 depicts the corresponding state diagram.

The iB2C behaviour network that is created from the FSM depicted in Fig. 3.14 using Algs. 3.1 and 3.2 is shown in Fig. 3.15.

In the network, the initiation is realised by the behaviour called *Init*, which gets active when the behaviour network shall be enabled. Hence, there are two ways in which the system can enter s_0 (“Waiting”):

1. **Initiation:** If *Init* gets active after having been inactive before, the input conditions of $(CBS) s_0$ will be fulfilled. As a result, $(CBS) s_0$ will get active.
2. **Returning to base:** The system is in state s_3 (“Driving to Base”), represented by $(F) s_3$ being active. If the activity of *Drive to Base* then falls from a value > 0 down to 0 and the activity of *Base Reached* rises to 1, the input conditions of $(CBS) s_{30}$ are fulfilled. As a result, $(CBS) s_{30}$ will get active.

According to Alg. 3.1 (see Lines 26 ff.), s_0 is therefore represented by a fusion behaviour $((F) s_0)$ that combines $(CBS) s_0$ and $(CBS) s_{30}$ instead of by a single CBS. As can be seen in Fig. 3.14, s_3 (“Driving to Base”) can also be reached via two transitions:

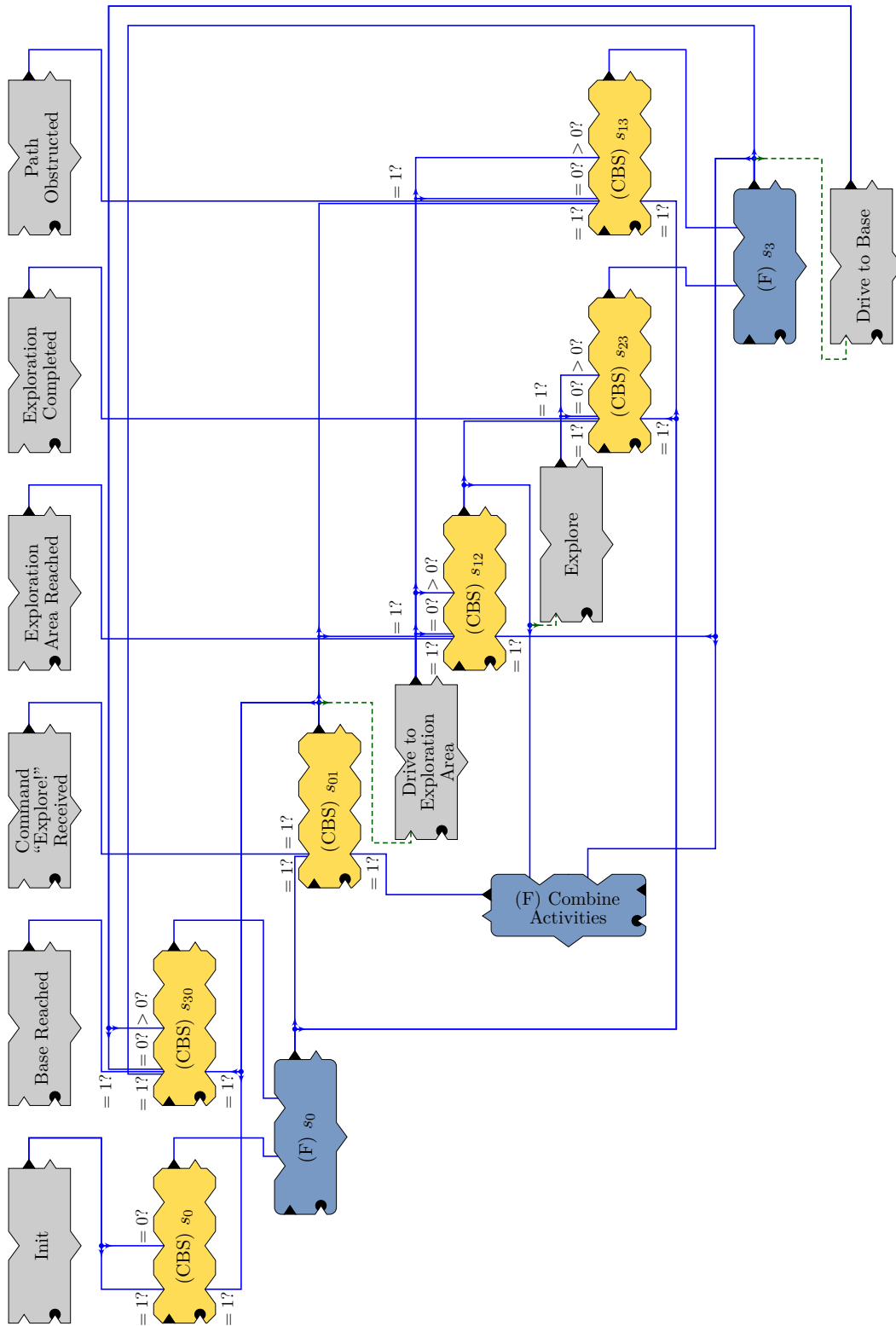


Figure 3.15: The behaviour network realising the complex task described by the FSM depicted in Fig. 3.14. A black triangle in a stimulation port means that the behaviour is always stimulated. Behaviours without stimulation input are stimulated by behaviours that are not part of the depicted network.

1. **Failure to reach exploration area:** The system is in state s_1 (“Driving to Exploration Area”), represented by $(CBS) s_{01}$ being active. If the activity of *Drive to Exploration Area* then falls from a value > 0 down to 0 and the activity of *Path Obstructed* rises to 1, the input conditions of $(CBS) s_{13}$ are fulfilled. As a result, $(CBS) s_{13}$ will get active.
2. **Completion of exploration:** The system is in state s_2 (“Exploring”), represented by $(CBS) s_{12}$ being active. If the activity of *Explore* then falls from a value > 0 down to 0 and the activity of *Exploration Completed* rises to 1, the input conditions of $(CBS) s_{23}$ are fulfilled. As a result, $(CBS) s_{23}$ will get active.

If $(CBS) s_{13}$ or $(CBS) s_{23}$ gets active, so does $(F) s_3$. This corresponds to the system being in state s_3 (“Driving to Base”). There is one state that has two successor states, i.e. two outgoing transitions: s_1 (“Driving to Exploration Area”). The following cases yield a transition from s_1 to a successor state:

1. **Success in reaching exploration area:** As the system is in state s_1 (“Driving to Exploration Area”), $(CBS) s_{01}$ is active. If the activity of *Drive to Exploration Area* then falls from a value > 0 down to 0 and the activity of *Exploration Area Reached* rises to 1, the input conditions of $(CBS) s_{12}$ are fulfilled. As a result, $(CBS) s_{12}$ will get active, indicating that the system has transitioned to s_2 (“Exploring”).
2. **Failure to reach exploration area:** As the system is in state s_1 (“Driving to Exploration Area”), $(CBS) s_{01}$ is active. If the activity of *Drive to Exploration Area* then falls from a value > 0 down to 0 and the activity of *Path Obstructed* rises to 1, the input conditions of $(CBS) s_{13}$ are fulfilled. As a result, $(CBS) s_{13}$ will get active, indicating that the system has transitioned to s_3 (“Driving to Base”).

According to Lines 51 ff. of Alg. 3.2, the activity output of the behaviour corresponding to a state is connected to a port associated with a feedback condition at each of the behaviours corresponding to the predecessor states. Depending on whether the state in question has only one or multiple predecessors, the behaviour corresponding to it is a CBS node or a fusion behaviour, which has to be taken into account when drawing the connection (cp. Lines 54 and 57). As s_1 (“Driving to Exploration Area”) has two successor states (s_2 and s_3), two behaviours ($(CBS) s_{12}$ and $(F) s_3$) have to be connected in the described way to the behaviour corresponding to s_1 ($(CBS) s_{01}$). Connecting both behaviours directly to feedback condition ports of $(CBS) s_{01}$ would not produce the desired result as the feedback conditions of a CBS node are connected in a logical conjunction (connection via logical AND). However, what is needed here is a logical disjunction (connection via logical OR). Hence, the fusion behaviour $(F) Combine Activities$ realises a combination of the activities of $(CBS) s_{12}$ (cp. Line 68) and $(F) s_3$ (cp. Line 71) via a logical OR and forwards the result to the feedback conditions port of $(CBS) s_{01}$ (cp. Line 75).

As explained above, each condition is checked by a single behaviour (*Base Reached*, *Command “Explore!” Received*, *Exploration Area Reached*, *Exploration Completed*, and *Path Obstructed*). The activity outputs of these behaviours are connected to ports related to enabling input conditions of the respective CBS nodes. Similarly, for each subtask there is a behaviour (*Drive to Exploration Area*, *Explore*, and *Drive to Base*) that realises the

actions belonging to this task. Each of these behaviours is stimulated by the corresponding CBS or fusion behaviour. Subdividing functionalities in this way is again an example of the behaviour-based approach. The advantage is that during the implementation phase of the system, separate system specialists (see Fig. 3.13) can work on a number of behaviours simultaneously as described in Sec. 3.2.1. Furthermore, each of the behaviours can be replaced by an improved version easily as soon as one is available or can be integrated into another system where the same partial functionality is needed.

Realising the execution of a complex task as a network of interacting behaviours offers other parts of the network the possibility of influencing the task execution. Two modifications of the behaviour network that is depicted in Fig. 3.15 shall illustrate this. The resulting network is shown in Fig. 3.16, in which the modifications are marked with red rectangles. The behaviour *Explore*, which realises the actions needed for exploring an area, has been replaced with the behaviour group (see Sec. 2.2.1) *(G) Explore*. Such an action can be sensible in case the actions that have to be executed are so complex that they can be better realised by a sub-network. As iB2C behaviour groups have the same interface as single behaviours, the substitution of the behaviour group *(G) Explore* for the behaviour *Explore* does not affect the remainder of the network.

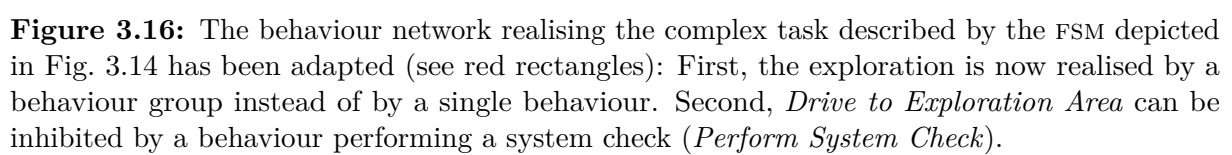
The second modification demonstrates how an external behaviour can influence a network realising a complex task. For this purpose, a new behaviour *Perform System Check* has been added. As the name suggests, its task is to perform a system check of the robot, for which the robot has to be stopped until the check has been completed. The activity of *Perform System Check* is high as long as the check is being performed and goes down to 0 as soon as it has been completed. Its activity output port is connected to the inhibition input port of *Drive to Exploration Area*. As a result, *Drive to Exploration Area* will be inhibited by *Perform System Check* whenever the latter gets active. This will prevent the robot from driving to the exploration area, which it should not do while a system check is being performed. During this time, the sub-network responsible for the exploration task will stay in state s_1 (“Driving to Exploration Area”).

There are other ways how external behaviours can interact with behaviours inside the sub-network. For example, an external behaviour could stimulate a behaviour within the network. In more complex cases, it will be sensible to extend the network in a way to respond to external influences. The advantage of such an interaction is the seamless integration of a sub-network realising a complex task into a surrounding behaviour network.

The extended example of a robot exploring an area has shown that the transformation of a rather simple Moore machine according to Algs. 3.1 and 3.2 yields an iB2C network with a considerable amount of behaviours. In the following section, a worst case estimation of the complexity in terms of the number of behaviours in a network corresponding to a given FSM is presented.

3.2.3 Complexity

The algorithm described in Algs. 3.1 and 3.2 consists of a number of (partially nested) loops that process several sets of elements. As the algorithm is only executed once for each Moore machine that shall be transferred into an iB2C network, its complexity is not relevant—neither in terms of time, nor in terms of space. What is relevant, however, is the complexity of the resulting network in terms of behaviours. This can, for example, be



used to estimate the size of the models needed for the verification concept described in Chap. 4. Thus, the following section will provide information about how the number of behaviours in an iB2C network created by the described algorithm depends on the Moore machine it was created from.

The following calculations are based on Def. 2.5, which defines a Moore machine as a 6-tuple $(S, s_I, \Sigma, \Lambda, T, G)$, with S being a set of states, $s_I \in S$ an initial state, Σ an input alphabet, Λ an output alphabet, $T : S \times \Sigma \rightarrow S$ a transition function, and $G : S \rightarrow \Lambda$ an output function. The FSM and the behaviour networks of the exploration task (see Sec. 3.2.2) are used to illustrate the calculations.

For each state $s \in S$, at most one behaviour has to be instantiated that shall execute the actions associated with this state s . This yields $|S|$ behaviours.

For each transition, at most one behaviour has to be instantiated that checks the condition associated with the transition. This yields a theoretic maximum of $|S \times \Sigma| = |S| \cdot |\Sigma|$ behaviours for the case that in each state $s \in S$ each input $\sigma \in \Sigma$ leads to a transition. Usually, the actual number will be much smaller as T can be a partial function and in each state s , only a subset of T will lead to a transition.

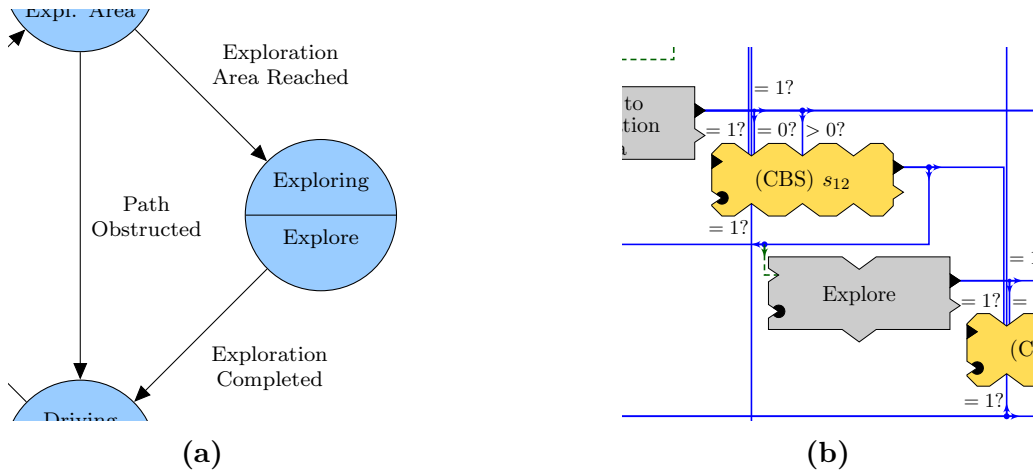


Figure 3.17: The state s_2 (“Exploring”) (see Fig. a) is represented by the CBS node $(CBS) s_{12}$ (see Fig. b).

For a simple FSM in which no state has more than one predecessor state, the number of behaviours representing states is $|S|$ because each state is represented by exactly one CBS. This case is illustrated by Fig. 3.17. However, more CBS nodes and a fusion behaviour are needed for each state s with more than one predecessor state, i.e. with more than one transition leading to s . This case is illustrated by Fig. 3.18. In the worst case there is more than one incoming transition in every state. Hence, for each of the $|S|$ states, a fusion behaviour is needed. Concerning the total number of transitions, the worst case is again that in each state $s \in S$ each input $\sigma \in \Sigma$ leads to a transition, which yields $|S| \times |\Sigma|$ transitions for which the same number of CBS nodes is needed. This yields $|S| + |S| \cdot |\Sigma|$ behaviours that are needed for representing states.

Finally, one fusion behaviour realising a connection via a logical OR has to be created for each state s with more than one successor state, i.e. with more than one transition leading away from it. This case is illustrated by Fig. 3.19. Here, the worst case is that every single

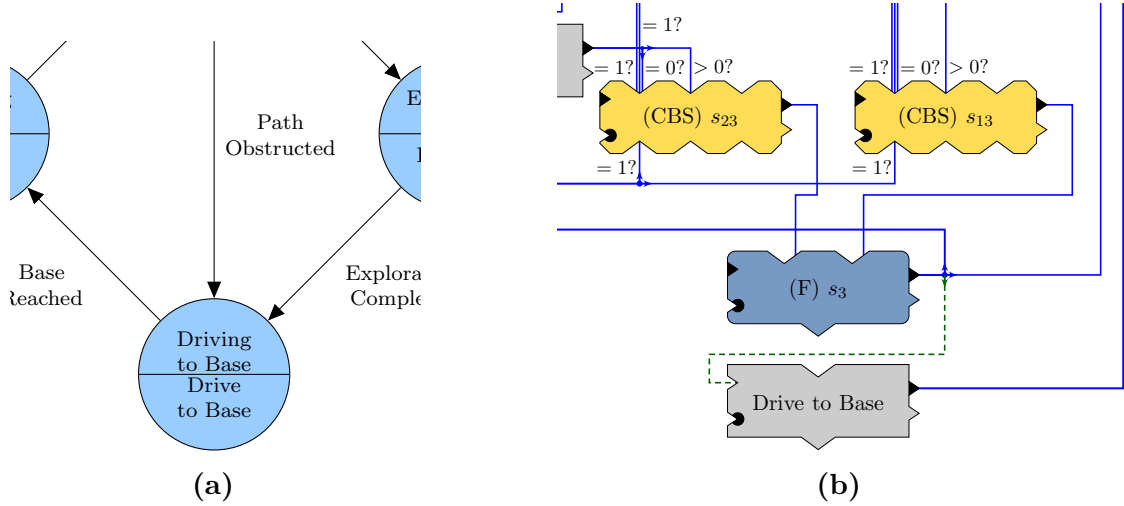


Figure 3.18: The state s_3 (“Driving to Base”) (see Fig. a) is represented by the fusion behaviour $(F) s_3$ (see Fig. b).

state $s \in S$ has more than one successor state. This necessitates the instantiation of $|S|$ fusion behaviours realising OR-connections.

In total, this means that at most $|S| + (|S| \cdot |\Sigma|) + (|S| + |S| \cdot |\Sigma|) + |S| = 2 \cdot (|S| \cdot |\Sigma|) + 3 \cdot |S|$ behaviours have to be instantiated for an iB2C behaviour network corresponding to the FSM $(S, s_I, \Sigma, \Lambda, T, G)$.

Apart from that, a behaviour may be needed for realising the initiation of the task execution. As explained above, this behaviour is connected to the CBS node representing the initial state $s_I \in S$. In case there is a transition leading to s_I , a fusion behaviour combining the CBS nodes associated with s_I is needed. Hence, at most three further behaviours have to be added to the number calculated above.

In the case of the extended exploration task illustrated in Figs. 3.14 (state diagram of Moore machine) and 3.15 (iB2C behaviour network), the above calculation yields a maximum number of $2 \cdot (|S| \cdot |\Sigma|) + 3 \cdot |S| = 2 \cdot (4 \cdot 5) + 3 \cdot 4 = 52$ behaviours, not including additional behaviours needed for the initiation. However, the network actually consists of only 15 behaviours (again neglecting the behaviours needed for the initiation).

The complexity analysis shows that the transformation of a small FSM can yield a network with a comparably large number of behaviours. In real-life applications, the number of necessary behaviours will typically be smaller. This is due to the fact that T is often a partial function, i.e. not all pairs (s, σ) with $s \in S$ and $\sigma \in \Sigma$ will result in a transition to a new state. Furthermore, the instantiation of behaviours checking conditions or executing actions as described in the algorithm is straightforward, but not optimised with respect to the total number of behaviours in the resulting network. It is possible to instantiate each behaviour only once and to connect it to different CBS nodes. That way, the modular structure of behaviour networks could be taken advantage of by using an instantiated behaviour in different places of the network.

3.2.4 Graphical Tool Support

Algorithms 3.1 and 3.2 describe how a Moore machine defining a complex task can be transformed into an iB2C behaviour network that realises said task. This algorithm is

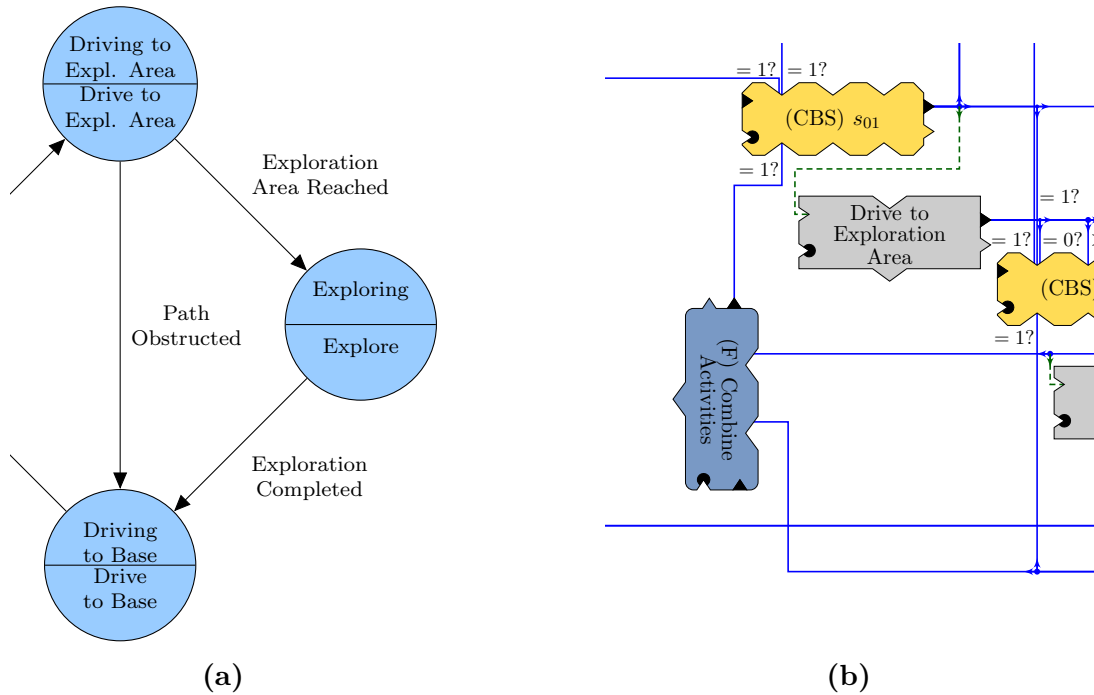


Figure 3.19: The state s_1 (“Driving to Exploration Area”) (see Fig. a) has two successors, which necessitates the instantiation of a fusion behaviour (see Fig. b).

of little help for the developer of a robot control system if there is no tool support that allows for specifying the Moore machine and automating the transformation process.

A FINSTRUCT (see Sec. B.2) widget has been developed and implemented that allows for graphically defining Moore machines (see [Rohr 12]). It is depicted in Fig. 3.20. The user can switch between three input modes of the mouse using the three icons in the top right corner of the widget:

1. The *Editing Mouse Mode* allows for creating states and transitions.
2. The *Picking Mouse Mode* allows for moving states or groups of states.
3. The *Transforming Mouse Mode* allows for moving, rotating, or shearing the entire Moore machine.

Apart from the standard controls for altering the view (scrollbars and buttons for zooming in and out), the view allows for saving a Moore machine or loading a previously saved Moore machine. It is also possible to export the Moore machine as an image file. The widget depicted in Fig. 3.20 shows a Moore machine defining the extended exploration task.

The algorithm consisting of Algs. 3.1 and 3.2 has been implemented and integrated into FINSTRUCT (see [Rohr 13]). It can be called directly from the above-mentioned widget. When called, it will convert the previously designed Moore machine into a corresponding network of iB2C behaviours, stored in the XML format of FINROC networks. FINROC programs can interpret this file, instantiate the corresponding behaviours, and connect

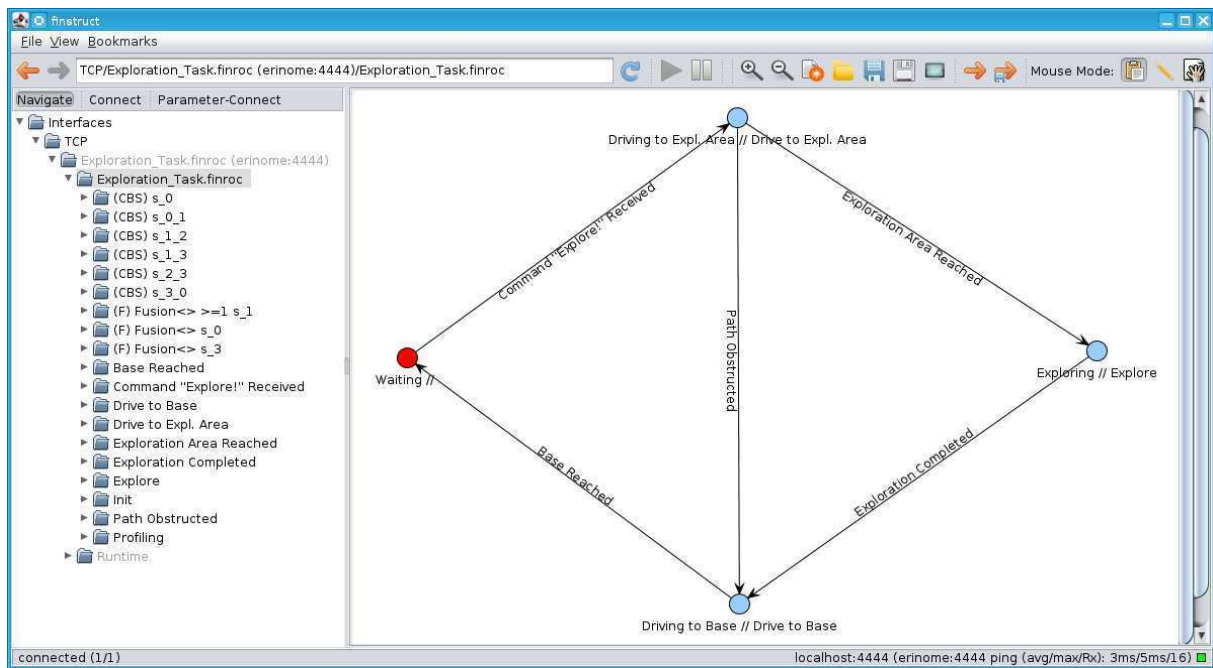


Figure 3.20: The FINSTRUCT widget that allows for graphically defining Moore machines. It depicts the Moore machine defining the extended exploration task. The initial state is marked in red, while the text below a state indicates its name and the outputs (actions) in the respective state, separated by a double slash. The edges are annotated with the conditions of the corresponding transitions. The automatic transformation of the Moore machine into a corresponding iB2C network can be directly initiated from this widget.

them appropriately. The result is the output of the automatic mapping as depicted in Fig. 3.13 and described in Sec. 3.2.1.

The iB2C network resulting from the automatic transformation of the Moore machine depicted in Fig. 3.20 can be seen in Fig. 3.21. It is visualised by the iB2C widget of FINSTRUCT, which allows for displaying iB2C networks realised as FINROC groups. Standard behaviours are visualised with grey shapes, while blue shapes represent fusion behaviours. The orange shapes depict CBS nodes. The horizontal bars inside a behaviour node indicate the behaviour's activation (yellow), activity (green), and target rating (red), respectively. As can be seen, the initiating behaviour is currently active. It had been inactive before, thus the input conditions of $(CBS) s_0$ are fulfilled. Hence, the latter is active, resulting in $(F) s_0$ also being active. Connections between behaviours are represented by black arrows. By clicking on an arrow, more information about the connection is provided.

Using the iB2C widget of FINSTRUCT, a developer can analyse a running system and directly see in which state it is (indicated by the activities of the CBS nodes and fusion behaviours representing states), why it is in that particular state (indicated by the activities of behaviours forming input and feedback conditions), and what the system is currently trying to achieve (indicated by the activities of the behaviours working on subtasks).

Graphical interfaces to robot control systems are common and described in a large variety in the literature. In [Bohren 11], a Python library called SMACH is described that can be used to build and execute hierarchical concurrent state machines. A SMACH-ROS⁴ interface library provides a graphical interface for runtime introspection of a system. The interface, however, does not allow for graphically specifying a system as a finite-state machine and then automatically creating a behaviour network realising this machine. This automatic generation along with the ability to visually inspect a system is one of the strengths of the approach presented here.

3.2.5 Discussion

In this section, a process for transferring a Moore automaton that defines a complex task into an iB2C behaviour network that realises the task has been introduced (see Sec. 3.2.1). An algorithm has been presented that can be used to automatically transfer the structure of the Moore machine into the skeleton of an iB2C network. In doing so, a sophisticated robot control system realising complex tasks can be created that is entirely based on a behaviour architecture. As stated by Design Decision 3, the encoding of the task has been realised in a decentralised way using special inter-behaviour connections.

The example that has been presented in Sec. 3.2.2 is a simple task that can be represented by an FSM of only four states. In Chap. 5, a much more complex real-world example will be presented. The complexity analysis of Sec. 3.2.3 has shown that the theoretically maximum number of behaviours in the resulting networks can be large. As has been explained there, the actual number is likely to be smaller due to the actual structure of the FSM and due to optimisation steps. However, it can be estimated that a network realising a complex task which has been designed following the proposed concept will have numerous behaviours.

⁴ROS: Robot Operating System; website: <http://www.ros.org/>

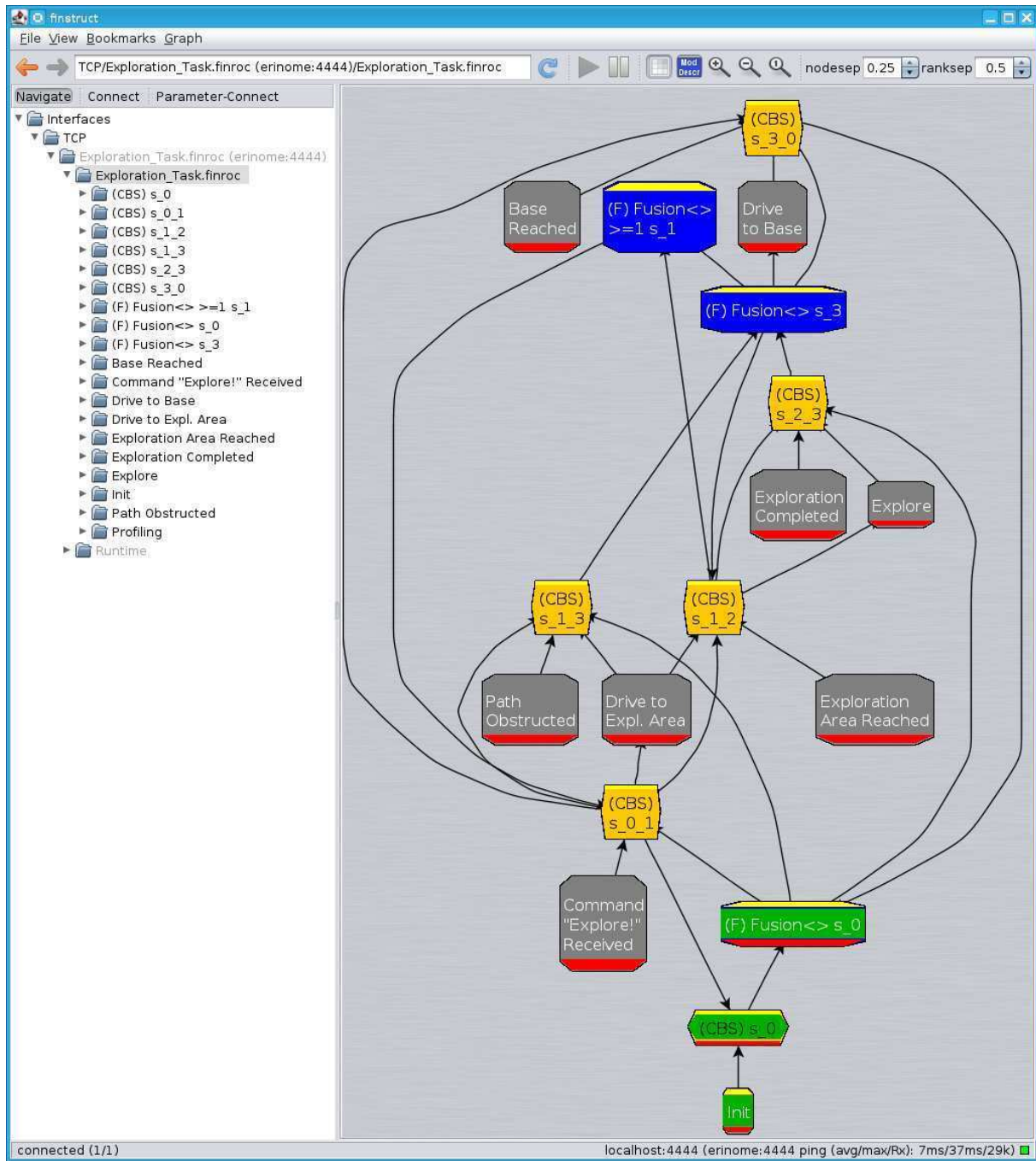


Figure 3.21: The FINSTRUCT iB2C widget that allows for graphically displaying iB2C networks that have been implemented as FINROC groups. The widget depicts the network that has been automatically created from the Moore machine shown in Fig. 3.20. Standard behaviours are visualised with grey shapes, while blue shapes represent fusion behaviours and orange shapes depict CBS nodes. The horizontal bars inside a behaviour node indicate the behaviour's activation (yellow), activity (green), and target rating (red), respectively.

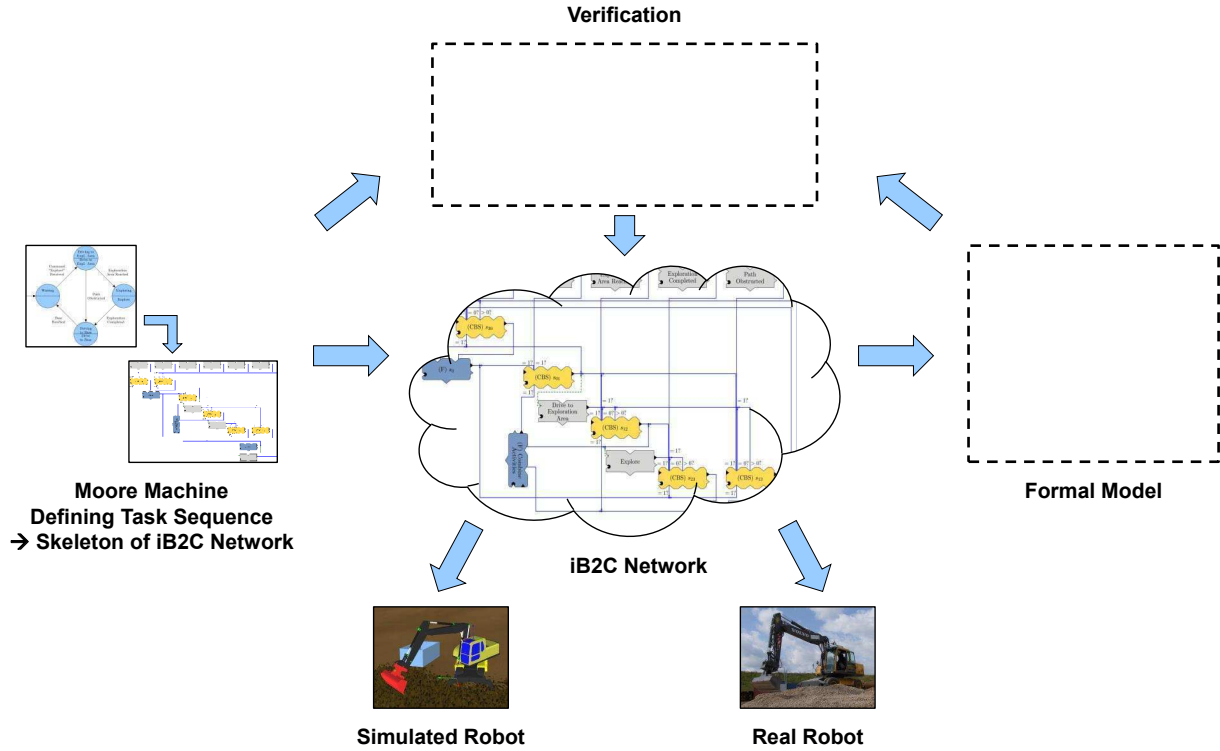


Figure 3.22: The proposed concept for encoding task sequences in iB2C networks has been integrated into the overall concept of this doctoral thesis.

In Sec. 3.2.4, tool support for the creation of such networks has been described. This support greatly facilitates the development of behaviour networks realising task sequences and thus is in line with Design Decision 4. However, the more complex a behaviour network gets and the more components are added after the initial automatic creation using said tool support, the more difficult it is to determine the actual functionality of the resulting network and to discover possible side-effects of the interaction of the behaviours. How this problem can be approached using verification techniques is the topic of Chap. 4.

Figure 3.22 depicts how the concept for encoding task sequences in iB2C networks presented in this chapter has been incorporated into the overall concept of this doctoral thesis.

4. Verifying iB2C Networks

In the previous chapter, a concept for encoding task sequences in iB2C networks has been presented. The networks that result from this encoding tend to consist of numerous behaviours with a large number of interconnections. While this is typical of BBS in general, the fact that sequences are encoded into behaviour networks further increases the numbers of behaviours and connections.

With tools like FINSTRUCT, it is easy to see in which state a system is. Assessing the overall functionality of a BBS or finding undesired side-effects is, however, much more difficult. Even if a BBS has been created from a formal description of a task using an automatic transformation as described in Sec. 3.2, the later change of the BBS or the addition of further behaviours can create errors in the system that may eventually lead to an undesired overall behaviour.

The most reliable solution to such problems is the application of formal verification techniques. These allow for checking whether a BBS fulfils certain given properties or whether certain (undesirable) states can be reached. For the work at hand and in particular with respect to task sequences, one class of questions is of special interest with regard to the verification of a BBS: Under which circumstances does a robot execute a task? In terms of behaviour-based systems, this question can be formulated as follows: Under which circumstances can a behaviour get active?

A concrete example of this type of question is the following: Assuming that the navigation system of a robot like RAVON (see Fig. 2.20 and Sec. 3.1.2) has several components that try to achieve a task, namely to navigate the robot in a specific way, under which circumstances is each of these tasks executed? In terms of BBS, the question can be put in the following way: Assuming that several behaviours in the behaviour-based part of RAVON's navigation system are able to realise the same task in different ways, under which circumstances can each of them get control over the robot, i.e. under which circumstances can each of them send its desired target coordinates to the lower control layers?

In this chapter, a verification technique will be introduced with which such questions can be answered. This is a prerequisite for the further analysis of iB2C networks realising task sequences.

There are several techniques for the formal verification of software components that have different advantages and disadvantages. Section 4.1 defines the term verification, gives a brief overview of selected verification techniques, and describes why model checking has been chosen for the work at hand.

In order to apply this technique to an iB2C behaviour network, the latter has to be transferred into a formal, simplified model. An approach to this is described in Sec. 4.2, which presents how iB2C behaviour networks can be modelled as networks of finite-state automata.

The actual verification using said networks of finite-state automata is presented in Sec. 4.3. In that section, it is described which properties of BBS can be verified and how tools can assist a developer in the verification process. The order of precedence of behaviours in RAVON’s navigation system is used to illustrate how verification can be performed.

4.1 Verification Techniques

The aim of this section is to define the term verification as it is used in the work at hand and to present two widely-used verification techniques that could be applied to the verification of iB2C behaviour networks.

In the literature, numerous definitions of the term “verification” can be found (e.g. in [IEEE 1012 04]). It is often not clearly distinguished from the related term “validation”, although many researchers consider the two terms as fundamentally different. In this thesis, the following definitions of the two terms are used. They are based on the definitions given in [Boehm 81].

Definition 4.1: Verification

The aim of *verification* (from Latin *veritas*, “truth”) is to establish the truth of correspondence between a robot control system and its specification.

Definition 4.2: Validation

The aim of *validation* (from Latin *valere*, “to be worth”) is to establish the fitness or worth of a robot control system for its mission.

[Boehm 81] is a publication from the area of software engineering. Accordingly, its author has described verification as answering the question “Are we building the product right?” and validation as responding to the question “Are we building the right product?”. The author of [Sommerville 11] shares this view and explains that the “aim of verification is to check that the software meets its stated functional and non-functional requirements”, while validation shall “ensure that the software meets the customer’s expectations”. This notion can be transferred to the area of robotics, where there is a big difference between the questions “Is the robot being built right?” and “Is the right robot being built?”.

For both verification and validation, there exist a number of methods. A system can be validated, for example, by simulation or testing. Both methods involve conducting experiments. Simulating a (robot control) system typically means building an abstraction or a model of it and checking its behaviour for different input values. Thus, the system’s

behaviour can be estimated without using the actual system. This is advantageous in case the manufacturing of the actual robot has not been completed yet or conducting experiments with the real system would be too dangerous. Similarly, the process of testing a system means checking its behaviour for different input values. The difference to simulation is that the real robot is used. While testing with a real machine often yields more realistic results than conducting experiments in simulation, results tend to be difficult to reproduce due to influences of the environment that cannot be controlled.

Simulation, as well as testing, is valuable for finding errors in a robot control system. Neither of the methods is, however, well-suited for proving that a control system complies with its specification, i.e. that it is error-free with respect to the specification. This could only be achieved by conducting experiments with all possible combinations of (classes of) input values, which is normally not feasible. As the computer scientist Edsger W. Dijkstra pointed out in [Buxton 70], testing “shows the presence, not the absence of bugs”. Therefore, verification techniques have to be applied in case a system shall be proven to be correct. A number of approaches to formal verification have been developed by now (see [McMillan 00] for a brief overview of verification techniques developed during several decades). In the following, deductive reasoning (see Sec. 4.1.1) and model checking (see Sec. 4.1.2) will be introduced as prominent examples of verification techniques.

4.1.1 Deductive Reasoning

The approach of using deductive reasoning for program verification goes back to works of the computer scientists Robert W Floyd (see [Floyd 67]) and C.A.R. Hoare (see [Hoare 69]). The basic idea is to define a *calculus* (a set of *rules*) and use *deduction* in order to prove the *partial correctness* of a program. Formally, the specification of a program can be defined by its *precondition* (input states that are relevant to the program¹) and its *postcondition* (property to be held after the execution of the program). Proving that a program is partially correct then means proving that it is correct with respect to its specification, in other words that the program’s postcondition holds after its execution in a state fulfilling its precondition. Furthermore, *total correctness* can be proven by proving that a program is partially correct and terminates.

In [Hoare 69], C.A.R. Hoare describes a formal system (later called “Hoare’s calculus” or “Hoare rules”) for proving the properties of programs. Hoare’s calculus can be used in two ways in order to prove partial correctness: as a deductive calculus or as a reduction calculus. When using it as a deductive calculus, the verification process starts with a set of formulae and the partial correctness assertions of two rules of the calculus. Using the other rules of the calculus, more complicated correctness assertions can be derived. In the end, this process should yield the assertion to be proven. When using Hoare’s calculus as a reduction calculus, on the contrary, the verification process starts with the assertion to be shown and successively tries to derive simpler assertions. The authors of [Sperschneider 91] argue that the reduction approach is far more efficient.

Examples of how to prove the partial correctness of a program using deductive reasoning in Hoare’s calculus can be found in [Sperschneider 91], [Sperschneider 96] (German), and [Nebel 12] (also German). In [Liu 02], a Hoare-style proof system for partial correctness of

¹Note: The meaning of the term “precondition” here is different from the meaning in Sec. 2.2.2.2.

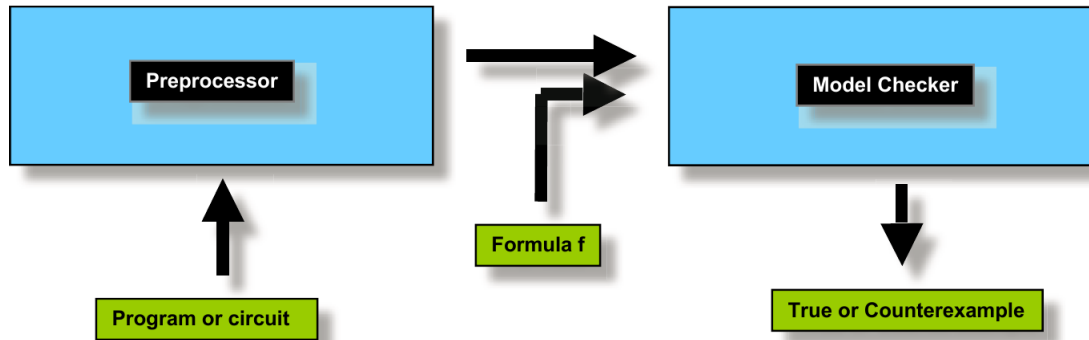


Figure 4.1: The concept of model checking (source: [Clarke 08]).

programs written in Golog, a logic programming language that can be used for high-level robot control, is presented.

There are a number of advantages of Hoare’s calculus (see [Sperschneider 91]), among them the fact that it has exactly one rule for each programming construct, which reduces the effort of program verification. But the verification of short and simple programs using Hoare’s calculus already shows that the verification process can get lengthy and tedious. The author of [Jalote 05] comments that much of the verification work has to be done manually, which can easily lead to clerical errors. In [Sperschneider 91], it is pointed out that Hoare’s calculus in the presented form is not suitable for verification and explained that there are more user-friendly versions. The presentation of these, however, is beyond the scope of the work at hand.

In the next section, an approach to automatic verification will be presented that offers some advantages over the approach of deductive reasoning.

4.1.2 Model Checking

The basic idea of what is today known as “model checking” has been developed independently by two groups of researchers in the early 1981s: Edmund M. Clarke and E. Allen Emerson in the USA (see [Clarke 82]) as well as J.-P. Queille and J. Sifakis in France (see [Queille 82])².

Figure 4.1 illustrates the different steps of model checking: A program or circuit to be analysed is transformed by a preprocessor into a state automaton. The property to be checked is provided as a formula of a temporal logic. The state automaton and the formula are given as input to a model checker, which yields as output whether the property holds or not. The figure also refers to a feature of many model checkers: providing a counterexample for a universal property that is *not* true or a witness for an existential property that *is* true (see comment about universal and existential properties below).

In the following, the term “model checking” is formally defined based on the definition provided in [Clarke 08].

²In [Clarke 08], Edmund M. Clarke discusses the question whether Amir Pnueli should be credited with inventing model checking because of his work published in [Pnueli 79].

Definition 4.3: Model Checking

Let M be a Kripke structure (i.e. a state-transition graph). Let f be a formula of temporal logic (i.e. the specification). *Model checking* is the process of finding all states s of M such that $M, s \models f$.

Hence, the word “model” in the term “model checking” refers to the question whether the temporal formula f is true in the Kripke structure M , i.e. whether M is a model for f . This definition of model checking uses special terms that shall be defined in the following. The first of these terms is “Kripke structures”. In simple words (see [Bérard 01]), “this is just another name for automata”. As formal definition, a slightly adapted version of the one given in [Clarke 99] shall be used:

Definition 4.4: Kripke Structure

A *Kripke structure* M over a set P_a of atomic propositions is a 4-tuple $M = (S, S_0, R, L)$ where

1. S is a finite set of states,
2. $S_0 \subseteq S$ is the set of initial states,
3. $R \subseteq S \times S$ is a transition relation that must be total, i.e. for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$, and
4. $L : S \rightarrow S^{P_a}$ is a function that labels each state with the set of atomic propositions which are true in that state.

The authors of [Clarke 99] comment that S_0 is omitted from the definition in cases where the initial states are not relevant, yielding $M = (S, R, L)$. The following definition of “temporal logic” is based on how the author of [Clarke 08] explains the term.

Definition 4.5: Temporal Logic

A *temporal logic* describes the ordering of events in time without introducing time explicitly. It extends predicate logic with special temporal operators. Temporal logics are often classified according to whether time is assumed to have a linear or a branching structure. The meaning of a temporal logic formula is determined with respect to a Kripke structure.

Information about predicate logic can be found in [Sperschneider 91], for example. There are a number of temporal logics, among which the *Computation Tree Logic* CTL* is widespread. Its formulae describe properties of *computation trees*, which can be built from a Kripke structure and contain all possible executions starting from the initial state. The building process is described in [Clarke 08]. In the CTL*, there are two *path quantifiers*, which are used to describe properties of computation paths or paths in the tree (f : formula):

1. Af (“all”): holds if and only if f holds for all computation paths

2. Ef (“exists”): holds if and only if f holds for at least one path

Properties of the form Af are called *universal properties*, while properties of the form Ef are called *existential properties*. Besides the path quantifiers, the CTL* offers the following five basic *temporal operators* (f, g : formulae):

1. Xf (“next time”): holds if and only if f holds in the next state of the path
2. Ff (“eventually”): holds if and only if f will hold in some state of the path
3. Gf (“always”): holds if and only if f holds in all states of the path
4. fUg (“until”): holds if and only if there is a state in which g holds and f holds in all preceding states of the path
5. fRg (“release”): holds if and only if g holds along the path up to and including the first state where f holds

The path quantifiers A and E are often combined with the temporal operators G and F . This yields the following four cases (f : formula):

1. AGf : holds if and only if f holds in all states of all paths
2. EGf : holds if and only if f holds in all states of some path
3. AFf : holds if and only if f holds in some state of each path
4. EFf : holds if and only if f holds in some state of some path

Temporal logics differ in the number and semantics of the temporal operators they provide. In other logics than CTL*, for example, A and E are written as \forall and \exists , while F and G are written as \diamond and \square (see [Ben-Ari 01]).

With regard to Def. 4.3, the actual process of verifying a program against its specification using model checking consists of the following three steps (see [Clarke 99]):

1. **Modelling:** The system to be verified has to be converted into a formal model that can be used as input to a model checker.
2. **Specifying:** The properties that the system shall have need to be specified formally (as formulae of temporal logic).
3. **Verifying:** Using the formal model of the system and the formal specification of its properties as input, the model checker evaluates whether the system conforms to its specification.

Model checking is a verification technique with widespread use that has already been applied to numerous applications. For example, the authors of [Scherer 05] describe how to use a model checker to verify the source code of a small line-following robot. The reason for the popularity of model checking is that it offers a number of advantages over other techniques. The author of [Clarke 08] provides a list of advantages and also explains the deficiencies and challenges of model checking. For the work at hand, the most relevant are the high degree of automation and the generation of witnesses and counterexamples, respectively. Therefore, model checking has been selected for the work described in this thesis as approach to verifying iB2C behaviour networks. The large state space is the major problem of the model checking approach. Its effect on the work at hand will be discussed in Sec. 5.2.

In the following, some information about the representation of Kripke structures will be provided.

4.1.2.1 Representations of Kripke Structures

The representation of the Kripke structure $M = (S, R, L)$ (see Def. 4.4) is crucial to the model checking process. In the first model checkers, the Kripke structure was represented as a labelled, directed graph with arcs given by pointers. The nodes of the graph represented the states S , its arcs defined the transition relation R , and its labels described the atomic propositions of L (see [Clarke 99]). This way of representing M is known as *explicit representation* and the model checking based on it as *explicit model checking*.

The major problem of model checking is that the number of states can quickly get very large. This is the so-called *state explosion problem*. It is tackled by representing large sets of states in a concise manner. This can be done by using a *symbolic representation* of the Kripke structure, which is the basis for *symbolic model checking*.

A widespread symbolic representation of Kripke structures is based on so-called ordered binary decision diagrams. They shall be presented briefly in the following, starting with binary decision trees (see [Clarke 99]).

Definition 4.6: Binary Decision Tree

A *binary decision tree* is a rooted, directed tree that consists of two types of vertices—terminal vertices and non-terminal vertices.

- Each non-terminal vertex v is labelled by a variable $\text{var}(v)$ and has two successors: $\text{low}(v)$ corresponding to the case where the variable v is assigned 0, and $\text{high}(v)$ corresponding to the case where v is assigned 1.
- Each terminal vertex v is labelled by $\text{value}(v)$, which is either 0 or 1.

Figure 4.2 depicts the binary decision tree that represents the two-bit comparator function of two variables $x = x_1x_0$ and $y = y_1y_0$, which is defined by the following formula:

$$f(x_0, x_1, y_0, y_1) = (x_0 \leftrightarrow y_0) \wedge (x_1 \leftrightarrow y_1)$$

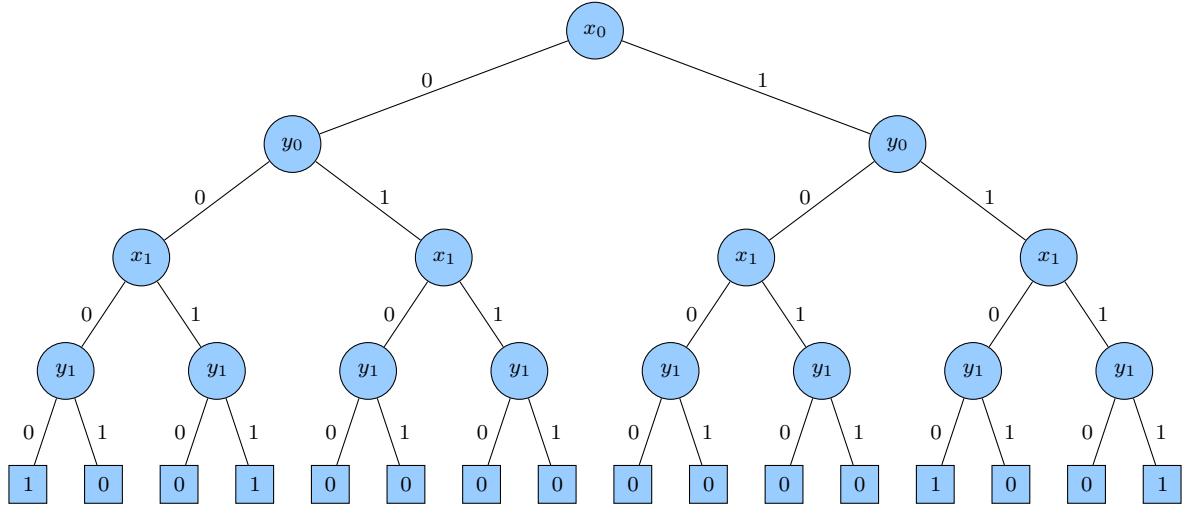


Figure 4.2: A binary decision tree that represents the two-bit comparator function of two variables $x = x_1x_0$ and $y = y_1y_0$ (cp. [Clarke 99]).

The value of f (true or false) for a given assignment can be determined by traversing the tree from its root to a terminal vertex and by selecting the successor of a vertex v depending on whether $\text{var}(v)$ is assigned 0 or 1.

The disadvantage of binary decision trees is that their structure is not optimised in any way. Hence, these trees are not a very concise representation of a formula. However, there is usually a lot of redundancy in these trees, which can be exploited in order to create a more compact representation: binary decision diagrams (see [Clarke 99]).

Definition 4.7: Binary Decision Diagram

A *binary decision diagram* (BDD) is a rooted, directed acyclic graph that consists of two types of vertices—terminal vertices and non-terminal vertices.

- Each non-terminal vertex v is labelled by a variable $\text{var}(v)$ and has two successors: $\text{low}(v)$ corresponding to the case where the variable v is assigned 0, and $\text{high}(v)$ corresponding to the case where v is assigned 1.
- Each terminal vertex v is labelled by $\text{value}(v)$, which is either 0 or 1.

The evaluation of $f_v(x_0, \dots, x_{n-1})$ defined by a BDD with root vertex v can be done with the following steps:

1. If v is a terminal vertex:

- (a) If $\text{value}(v) = 1$, then $f_v(x_0, \dots, x_{n-1}) = 1$
- (b) If $\text{value}(v) = 0$, then $f_v(x_0, \dots, x_{n-1}) = 0$

2. If v is a non-terminal vertex with $\text{var}(v) = x_i$:

$$f_v(x_0, \dots, x_{n-1}) = (\neg x_i \wedge f_{\text{low}(v)}(x_0, \dots, x_{n-1})) \vee (x_i \wedge f_{\text{high}(v)}(x_0, \dots, x_{n-1}))$$

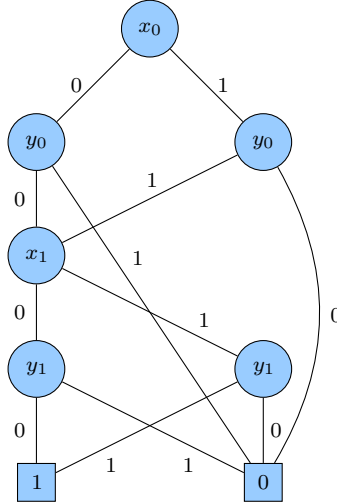


Figure 4.3: An ordered binary decision diagram that represents the two-bit comparator function of two variables $x = x_1x_0$ and $y = y_1y_0$. The order of the variables is $x_0 < y_0 < x_1 < y_1$ (cp. [Clarke 99]).

Based on Def. 4.7, a special type of BDD can be defined:

Definition 4.8: Ordered Binary Decision Diagram

An *ordered binary decision diagram* (OBDD) is a BDD in which the variables appear in the same order along each path from the root to a terminal vertex.

Figure 4.3 depicts the OBDD representing the above-mentioned two-bit comparator function with the variable ordering $x_0 < y_0 < x_1 < y_1$. As can be seen by comparing Figs. 4.2 and 4.3, the OBDD representation of the two-bit comparator has less vertices (8) than the representation as a binary decision tree (31). However, the ordering of the variables in an OBDD has a strong effect on its size. Moreover, finding an optimal ordering is in general infeasible according to [Clarke 99]. Therefore, heuristic methods are employed.

Using OBDDs in model checking allows for concisely representing the model of a system that shall be analysed. This necessitates the encoding of the states S , transition relation R , and labelling function L of a Kripke structure $M = (S, R, L)$. Information about how this can be done is provided in [Clarke 99].

There are also other ways of representing the model of a system that can be used for model checking. For example, the model checking toolbox UPPAAL (see Sec. 4.1.2.2) uses so-called *difference bounded matrices* (DBMs) (see [Behrmann 02]). Furthermore, *clock difference diagrams* (CDDs) are employed, which are BDD-like structures (see [Larsen 99]).

In the following, the model checker UPPAAL will be described as it is a technical basis for the verification of iB2C behaviour networks described in this thesis.

4.1.2.2 The Model Checking Toolbox Uppaal

UPPAAL is a model checking toolbox targeting the verification of real-time systems. It is jointly developed by researchers at Uppsala University and Aalborg University. There are

several publications about UPPAAL, for example [Behrmann 04] and its updated version [Behrmann 06]. Systems to be verified with UPPAAL are modelled as networks of extended timed automata. These automata (without the UPPAAL extensions) are described in [Alur 90]. In short, a timed automaton is a finite-state machine (cp. Sec. 2.1.1) extended with clock variables. The following definition is based on the one given in [Behrmann 06]:

Definition 4.9: Timed Automaton

A *timed automaton* is a tuple (L, l_0, C, A, E, I) , where L is a set of locations, $l_0 \in L$ is the initial location, and C is a set of clocks. $B(C)$ is a set of conjunctions over simple conditions of the form $x \otimes y$ or $x - y \otimes c$, where $x, y \in C$, $c \in \mathbb{N}$, and $\otimes \in \{<, \leq, =, \geq, >\}$. Furthermore, A is a set of actions, co-actions, and the internal τ -action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard, and a set of clocks to be reset, and $I : L \rightarrow B(C)$ assigns invariants to locations.

This formal definition is extended by UPPAAL with a number of additional features. Each UPPAAL automaton is realised as a process, which is an instantiation of a template (see the list below). It consists of a number of *locations* with unique names that are connected via a set of *edges*. Edges can be labelled with (amongst others) *guards* (side-effect free Boolean expressions to determine whether an edge is enabled), *updates* (assignments), and channel-based *synchronisations* between automata (see comments about channels below). Of all the features with which UPPAAL extends the concept of timed automata, the following list only contains the ones relevant to this thesis. A complete list can be found in [Behrmann 06].

- **Templates:** definitions of automata with parameters of any type; used to create several processes with different parameter values from one automaton definition
- **Constants:** integers that cannot be modified; declared as `const name value`
- **Bounded integer variables:** integers with a minimum and a maximum value; declared as `int [min,max] name`
- **Binary synchronisation channels:** used to synchronise exactly two automata; declared as `chan c`; edge labelled with sending channel `c!` synchronises with exactly one edge labelled with `c?`; if several combinations exist, synchronisation pair is chosen non-deterministically
- **Broadcast channels:** used to synchronise arbitrary number of automata; declared as `broadcast chan c`; edge labelled with sending channel `c!` synchronises with arbitrary number of edges labelled with `c?`; any receiver able to synchronise has to do so; sender is not blocked in case of no receiver
- **Urgent locations:** time does not pass in state with urgent location
- **Committed locations:** time does not pass in state with committed location; outgoing edge of at least one of the committed locations must be involved in next transition

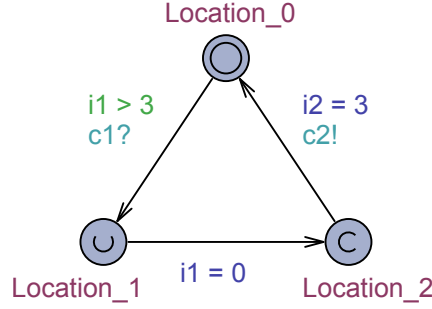


Figure 4.4: A simple automaton with three locations (Location_0: initial, Location_1: urgent, Location_2: committed), a guard ($i1 > 3$), two updates ($i1 = 0$, $i2 = 3$), and two channel synchronisations ($c1?$: receiver, $c2!$: sender).

- **Arrays:** allowed for clocks, channels, constants, and integers; declared as `type name[number]`
- **Initialisers:** used to initialise (arrays of) integers
- **User functions:** defined either globally or locally in templates; similar to C++

In Fig. 4.4, an example of a UPPAAL automaton is shown. It features the main elements needed for the work described in this thesis.

The query language used in UPPAAL is a simplified version of the *Timed Computation Tree Logic* (TCTL), which is introduced in [Alur 93]. It supports *path formulae* as well as *state formulae*, but in contrast to TCTL does not allow the nesting of path formulae. As the names suggest, state formulae are used to describe individual states, while path formulae are used to evaluate paths of a run of a UPPAAL system (compare the remarks about path quantifiers and temporal operators above).

A state formula is an expression like $a_value == 1$ for an integer variable a_value , a test whether a process is in a certain location like `ActivityCalculation.Active` for a process `ActivityCalculation` containing a location `Active`, or the keyword `deadlock`, which is used to check whether a system has reached a deadlock. Path formulae are used to check whether a property of the following types holds (φ, ψ state formula):

- **Reachability property:** A reachability property checks whether φ is fulfilled in any reachable state, i.e. whether there is a path starting in the initial state such that φ is eventually fulfilled on that path. This is written as $E\Diamond\varphi$ or $E<>\varphi$.
- **Safety property:** A safety property checks whether φ is fulfilled in all reachable states ($A\Box\varphi$ or $A[]\varphi$) or whether there is a maximal path (infinite path or path with no outgoing transition in the last state) on which φ is always fulfilled ($E\Box\varphi$ or $E[]\varphi$).
- **Liveness property:** A liveness property checks whether φ is eventually satisfied ($A\Diamond\varphi$ or $A<>\varphi$) or whether φ leads to ψ , i.e. whenever φ is fulfilled, then ψ will also be fulfilled eventually ($\varphi \rightsquigarrow \psi$ or $\varphi \dashrightarrow \psi$).

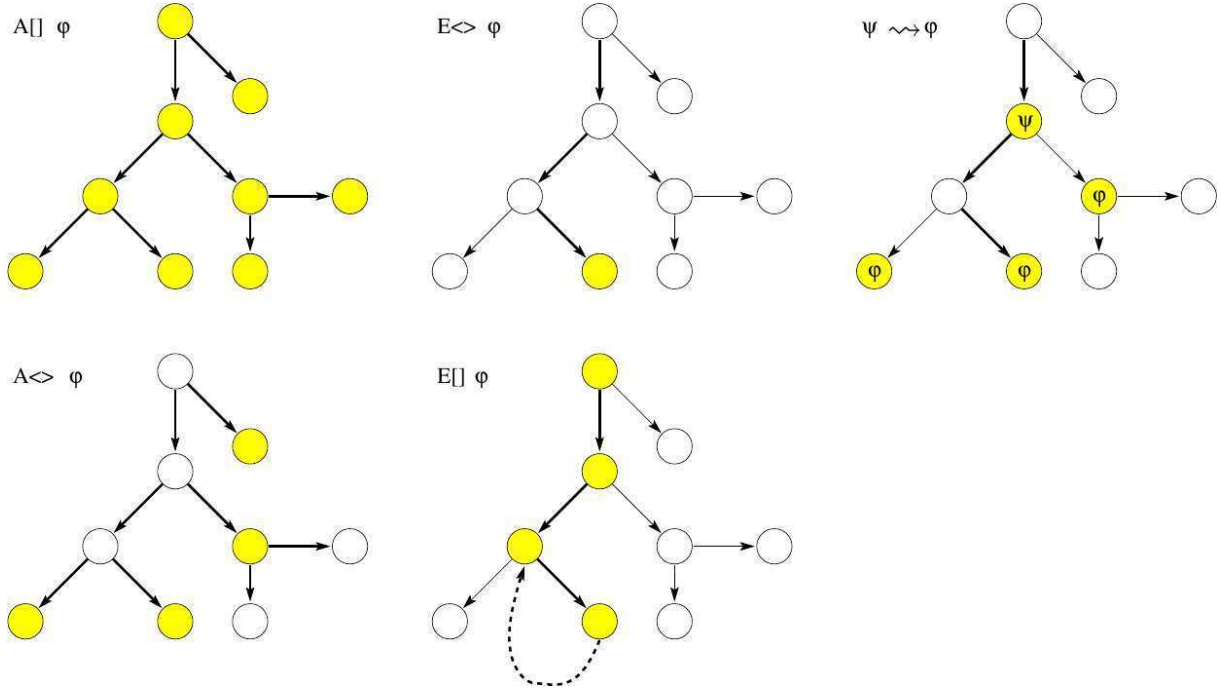


Figure 4.5: The different types of path formulae available in UPPAAL. The yellow states are the ones in which the given formula φ holds. According to [Behrmann 06], the bold edges depict the paths on which the formulae evaluate (source: [Behrmann 06]).

Figure 4.5 depicts the path formulae available in UPPAAL.

The graphical user interface of UPPAAL consists of three³ parts: the editor, the simulator, and the verifier. The editor is used to graphically define templates of timed automata and to add textual elements (e.g. variable declarations) to a system. With the simulator, it is possible to run a system—either by choosing transitions manually, by letting the simulator choose transitions randomly, or by going through a previously saved trace. In the verifier, finally, the user can enter queries that shall be used to check whether the given system fulfils certain properties. Figure 4.6 depicts a screenshot of the verifier that is in the process of checking a property of the behaviour network described in Sec. 5.2.2. The list in the upper part of the window in the background contains the loaded queries. Below, the query that is currently processed is shown. If a comment had been added to this query, it would also be displayed. At the bottom, general status messages are shown, which contain, for example, the results of the preceding verification processes. The window in the foreground provides detailed information about the current verification process, e.g. about the CPU time and the memory consumption.

The UPPAAL toolbox has been chosen for the work at hand as it offers a powerful graphical user interface, an XML-based file format for saving automata, and a stand-alone verifier that can be called from a command line. The latter two features are important for the automation of the verification process presented later in this work (see Secs. 4.2.2 and 4.3.2.3).

³Later versions of UPPAAL contain a fourth part, the so-called concrete simulator.

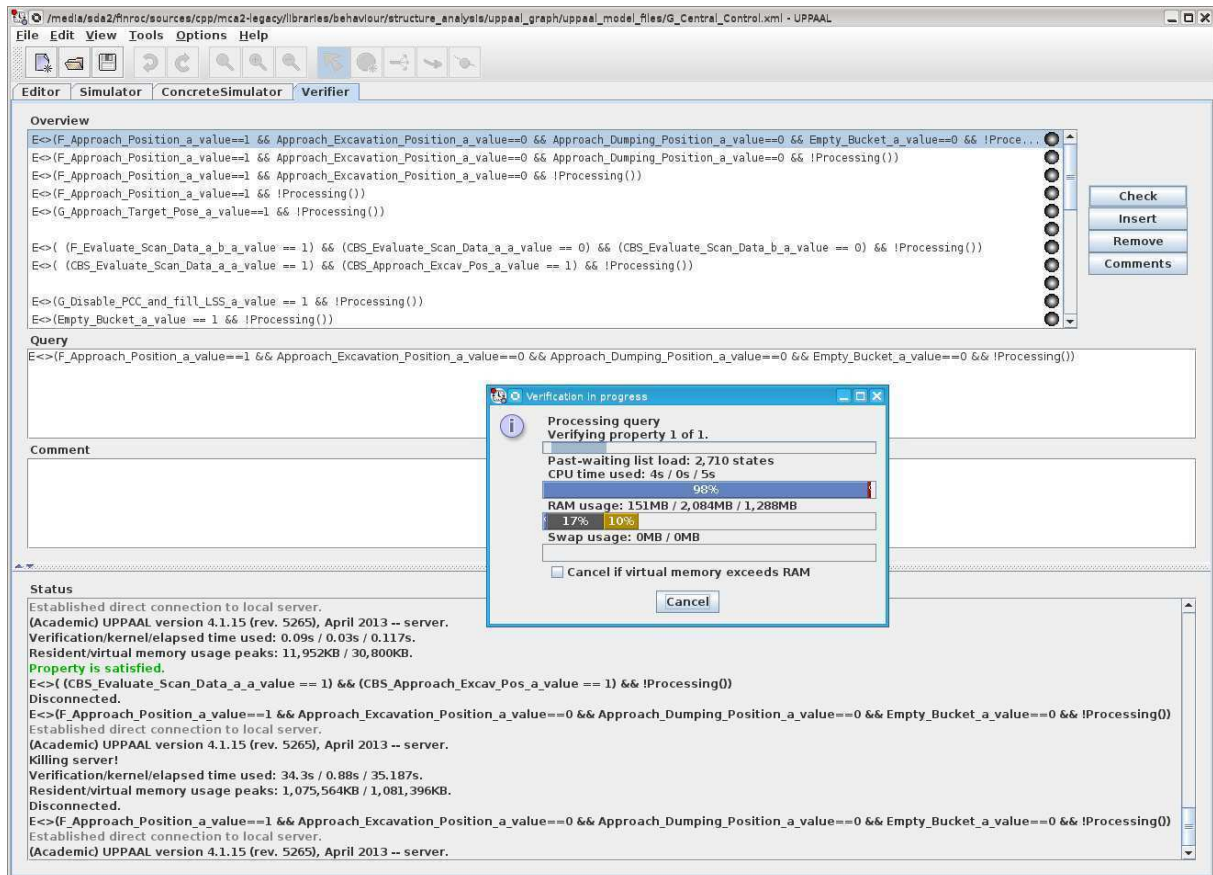


Figure 4.6: The verifier of UPPAAL's graphical user interface in the process of checking a property of a behaviour network. The window in the background provides information about the loaded queries and general status messages, while the window in the foreground displays detailed information about the current verification process.

The following sections will explain how the three steps of the model checking process (modelling, specifying, and verifying) have been realised in the context of the work described in this thesis. Section 4.2 will deal with modelling iB2C behaviour networks as networks of timed automata, while Sec. 4.3 will comment on how formal specifications can be formulated and how iB2C behaviour networks can be verified using said formal specifications.

4.2 Modelling iB2C Behaviour Networks as Networks of Uppaal Automata

In Sec. 4.1.2, model checking has been introduced as an approach to formal verification and it has been explained why model checking has been selected as the appropriate approach for the work at hand. As mentioned there, the general procedure of model checking consists of the following steps (see Fig. 4.1):

1. Create a state automaton as a model of the system to analyse.
2. Formulate the property to check as temporal formula.
3. Pass the state automaton and the temporal formula to a model checker, which will indicate whether the property holds or not.

In order to verify the correct operation of a behaviour network, these abstract steps have to be turned into a procedure that is specifically tailored to the type of behaviour network, the types of properties to be checked, and the features of the model checker that shall be used. In the case of iB2C behaviour networks, it has to be decided which aspects of the behaviours and their connections shall be modelled, how these aspects shall be represented in UPPAAL's timed automata, and how the modelling process shall be performed technically. The result is a number of design decisions, which will be explained in the following.

The first decision is what aspects to model. It has been explained several times that in BBS, a large part of the intelligence is encoded in the connections between the behaviours. Therefore, a sound model that shall be used as a basis for a verification process has to represent these connections. Furthermore, the behaviours themselves have to be mapped to a model consisting of automata. A feature of the iB2C is that its behaviours can get arbitrary complex as there is no limitation on their transfer function \vec{F} (see Sec. 2.2.1). This arbitrary complexity is difficult to model and can easily lead to very large automata. Large automata in turn will result in a huge state base for the verification process, which will make verifying the transfer function practically impossible. The authors of [Proetzsch 07] have represented floating point numbers as integers with a given bit width and have limited their experiments to behaviours with comparably simple transfer functions. In the work described in this thesis, a different path is taken: The transfer function \vec{F} of a behaviour is neglected, as are its input and output vectors \vec{e} and \vec{u} . At first sight, this decision might seem to be too radical. However, the interaction of iB2C behaviours depends heavily on the behaviour signals, i.e. stimulation s , inhibition i , activation ι , activity a , and target rating r . For example, orders of precedence between behaviours (see Sec. 4.3.1) can be

established using connections that transfer behaviour signals. In case such a connection is missing, an incorrect order of precedence may be established, which will result in the control system not working as expected. These considerations lead to the following design decision regarding the verification of iB2C behaviour networks:

Design Decision 5

The model of an iB2C behaviour network shall represent the calculation of the five behaviour signals s , i , ι , a , and r for each behaviour and the exchange of these signals between the behaviours.

After it has been determined *what* is to be modelled, the next question is *how* it can be modelled appropriately. All behaviour signals can take values from the interval $[0, 1]$. UPPAAL does not support floating point values, which is typical for a model checker. As a result, the value range of the behaviour signals has to be adapted. In many cases of verification, the degree with which a behaviour B is stimulated or inhibited is of little importance. What is important, is whether B is stimulated (or inhibited) *at all*. The same holds true for activity and target rating. For example, instead of asking “Can a_B rise above 0.75?”, it will be asked “Can B get active?”. In Secs. 4.3.1 and 5.2, it will be demonstrated that this is sufficient for real-world applications. Therefore, it is reasonable to restrict the value range of the behaviour signals to the set $\{0, 1\}$. With the restricted value range, the values of the behaviour signals can be represented as bounded integer variables in UPPAAL, which reduces the memory consumption and has the further advantage that UPPAAL can check for a violation of the bounds during verification. The result is the next design decision regarding the verification of iB2C behaviour networks:

Design Decision 6

The value of each of the five iB2C behaviour signals s , i , ι , a , and r shall be represented by a bounded integer with lower bound 0 and upper bound 1.

A further question is how many automata shall represent a behaviour network. Using a large number of simple automata has the advantage that each of them only has a small number of locations. But for more automata, the overhead for synchronising them is higher. In theory, it would be possible to map a complete iB2C behaviour network to only one automaton. While this automaton would not need any external synchronisation, it would consist of a huge number of locations for even a small behaviour network, which would make it difficult to design and hard to use during the verification process. Therefore, the approach that has been chosen for the work at hand models each behaviour as a fixed set of interconnected automata that depends on the type of the behaviour in question. What this exactly means is explained further below (see Sec. 4.2.1). The whole behaviour network is then modelled as a network of several sets of automata, one set for each behaviour in the network. This yields the next design decision regarding the verification of iB2C behaviour networks:

Design Decision 7

Each iB2C behaviour shall be modelled as a fixed set of interconnected automata that depends on the type of the behaviour. An iB2C behaviour network shall be represented by a network of such sets of automata in which there is a set of automata for each behaviour in the network.

Finally, it has to be decided how the modelling process shall proceed technically. Naturally, the *manual* creation of a network of automata from an iB2C network could only be done with reasonable effort for very small behaviour networks. In addition, the risk of making mistakes during the creation of the model would be high. Therefore, an *automatic* translation of an iB2C network into a network of automata that can be processed by UPPAAL is the method of choice. This leads to another design decision regarding the verification of iB2C behaviour networks:

Design Decision 8

The creation of a network of UPPAAL automata as a model of an iB2C network shall be realised in an automatic mapping process.

A significant difference to the work described in [Proetzsch 07] is that the model is built from the *existing* system here. By contrast, the authors of [Proetzsch 07] have reimplemented the behaviours to be verified in the synchronous language Quartz (see Sec. 2.2.1). The Quartz code was then exported to C code and reintegrated into the network in order to ensure consistency between the model and the actual code. This double work is unnecessary here as the model is created from the existing system.

In the following, it will be described in detail how the above design decisions have been implemented.

4.2.1 Mapping Standard iB2C Behaviours to Uppaal Automata

The five iB2C behaviour signals stimulation s , inhibition i , activation ι , activity a , and target rating r play a major role for the interaction of behaviours. Therefore, it has been decided that their calculation for each behaviour as well as their exchange between behaviours shall be covered by the automata-based verification (see Design Decision 5). It seems reasonable to dedicate one automaton to each of the five signals as this will result in automata of moderate complexity. By contrast, templates dealing with more than one behaviour signal would consist of much more locations and edges. This would decrease clarity and thus complicate enhancing or otherwise altering templates. Moreover, the chosen approach yields a good encapsulation, which facilitates the reuse of some templates for the models of different types of behaviours (cp. Secs. 4.2.3 and 4.2.4).

The five automata representing one standard behaviour B constitute one of the sets of automata requested in Design Decision 7. They will be described in detail in the following (cp. [Armbrust 12a]). “Standard” in this context refers to behaviours that are not modelled in a special way, in contrast to the two coordination behaviours presented in Secs. 4.2.3 and 4.2.4, for example. The five automata have been given the telling names `StimulationInterface`, `InhibitionInterface`, `ActivationCalculation`, `ActivityCalculation`, and `TargetRatingCalculation`.

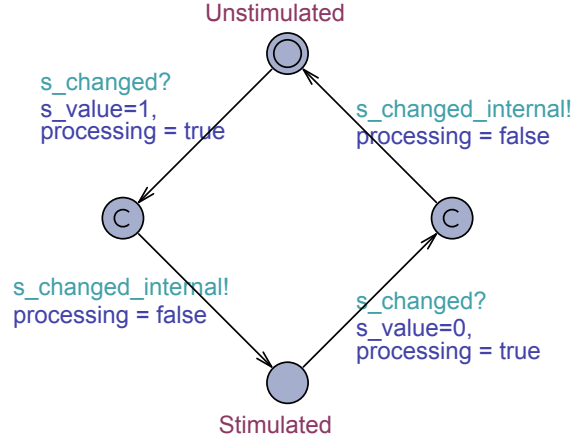


Figure 4.7: The `StimulationInterface` of a standard behaviour.

The `StimulationInterface` encapsulates the stimulation of a behaviour. It is depicted in Fig. 4.7. As can be seen, the automaton contains four locations and four edges connecting them. The main locations are labelled with `Unstimulated` and `Stimulated`, indicating that they represent that B is not stimulated ($s_B = 0$) or fully stimulated ($s_B = 1$), respectively. The representation of the stimulation as a binary value is in accordance with Design Decision 6. A change of the stimulation of B can be signalled by another automaton (see Sec. 4.2.2) using the channel `s_changed` (see information about channel-based synchronisation in Sec. 4.1.2.2); depending on the current location, the internal value of the stimulation (`s_value`) is then changed to 1 (currently at `Unstimulated`) or 0 (currently at `Stimulated`), respectively, and the automaton switches to `Stimulated` or `Unstimulated`, respectively. In both cases, the signal `s_changed_internal` is sent out. It is used for synchronising the `StimulationInterface` with the `ActivationCalculation` (see below). The transitions from `Unstimulated` to `Stimulated` and vice versa had to be split into two parts as UPPAAL does not support more than one synchronisation per edge. After the first synchronisation, the automaton enters a committed location, with the effect that a transition from this (or another) committed location has to be taken before a transition from a normal location can occur. This technique is applied in several automata. The use of intermediate locations is a technical workaround and does not add any further value to the model. In order to easily exclude committed locations from the results of queries (see Sec. 4.3), the flag `processing` has been added. It is set to true whenever the automaton switches to an intermediate location and back to false when it leaves the location. This workaround is used in other automata, too.

The automaton modelling the inhibitory inputs of a behaviour B —which is called `InhibitionInterface`—is built up in a similar fashion as the `StimulationInterface`: It receives information about a change of an inhibitory input via a channel, updates the variable storing the inhibition (`i_value`) if necessary, and sends an internal signal (`i_changed_internal`) if the inhibition has changed. Despite this similarity regarding the basic operation of the `StimulationInterface` and the `InhibitionInterface`, there is a significant difference concerning the creation of the automaton: While the stimulating input of a behaviour has dimension one, there can be an arbitrary number of inhibiting inputs (see the definition of the inhibition with $i = \|\vec{i}\|_\infty$ in Sec. 2.2.1). The

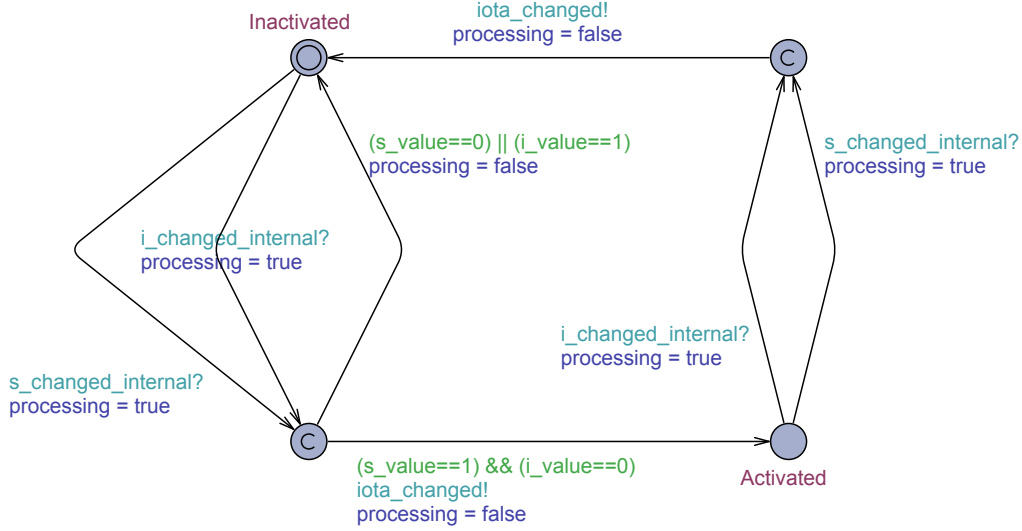


Figure 4.9: The ActivationCalculation of a single behaviour, calculating the behaviour’s activation from its stimulation and its inhibition.

InhibitionInterface must account for the number of inhibitory inputs of the corresponding behaviour. As a result, there is actually not only one **InhibitionInterface**, but a special template for each number of inhibitory inputs occurring in the iB2C network to be modelled. How these different templates are created is explained in Sec. 4.2.2. Figure 4.8 exemplarily depicts the **InhibitionInterface** of a behaviour B to which two inhibiting behaviours are connected. The main locations are **Uninhibited**, **Inhibited_by_First**, **Inhibited_by_Second**, and **Inhibited_by_Both**. In the initial location (**Uninhibited**), the automaton waits for a signal via one of the channels **i0_changed** or **i1_changed**, which models that the corresponding inhibitory behaviour has started to inhibit B . It then sets **i_value** to 1 and sends out the internal signal **i_changed_internal**, which is used for synchronising with the **ActivationCalculation**. As in the **StimulationInterface**, this transition had to be split up into two edges and an intermediate location due to the two synchronisations. After that, the automaton is in location **Inhibited_by_First** or **Inhibited_by_Second** depending on whether the inhibiting behaviour is connected to the first or the second inhibitory input port. If an inhibition by the remaining inhibitory behaviour is then signalled, the automaton will transition to location **Inhibited_by_Both**. In this case, **i_value** does not have to be updated, because the inhibition does not change. The way back to the location **Uninhibited** is straightforward. Again, the flag **processing** is provided to facilitate the exclusion of intermediate states when creating queries. The structure of the **InhibitionInterface** is that of a hypercube with dimension n_i if n_i inhibiting behaviours are connected to B . This facilitates the calculation of the numbers of locations and edges of the different variants of the **InhibitionInterface** (see Sec. 4.2.5).

The next automaton, **ActivationCalculation**, models the calculation of a behaviour’s activation based on its stimulation and inhibition. The general formula for this is $\iota = s \cdot (1 - i)$ (see again Sec. 2.2.1). In the UPPAAL models, the behaviour signals have been limited to a value range of $\{0, 1\}$. Therefore, the calculation can be replaced with the following rule: A behaviour B is only activated ($\iota_B = 1$) if it is stimulated ($s_B = 1$) and not inhibited ($i_B = 0$). For the verification approach taken here, this simple rule adequately

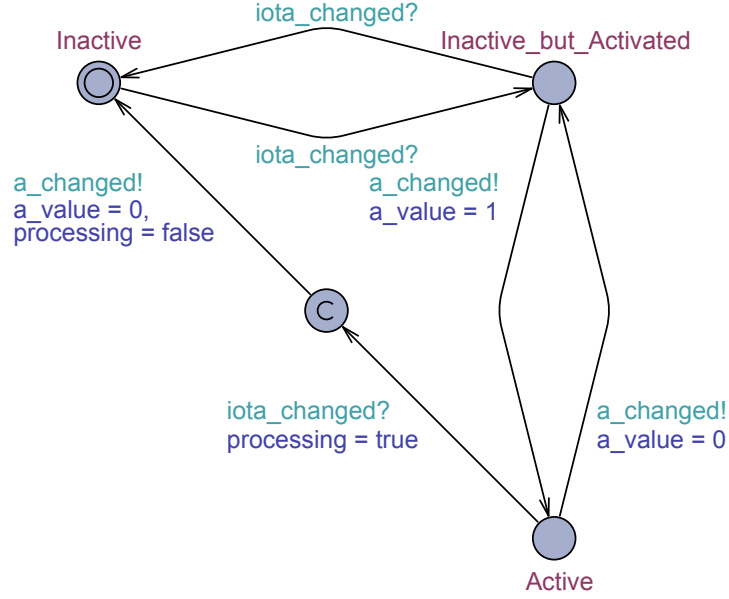


Figure 4.10: The ActivityCalculation of a standard behaviour, calculating the behaviour’s activity depending on its activation.

models the calculation of the activation in iB2C networks. In Fig. 4.9, the corresponding automaton is depicted. It possesses two main locations (*Inactivated* and *Activated*) modelling that B is *not* activated ($\iota_B = 0$) or activated ($\iota_B = 1$), respectively. The initial location is *Inactivated*. On receiving a signal via channel *s_changed_internal* or channel *i_changed_internal*, the *ActivationCalculation* transitions to the bottom left committed location. As has been described above, these two signals are sent out by the *StimulationInterface* and the *InhibitionInterface*, respectively, and realise the synchronisation between the three mentioned automata. Two edges lead away from the committed location—one back to the location *Inactivated* and one to the location *Activated*. Which one of them can be taken depends on the evaluation of two guards: If $s_value == 0$ or $i_value == 1$, then the edge back to *Inactivated* is taken. In the opposite case, i.e. $s_value == 1$ and $i_value == 0$, the other edge is taken, resulting in the automaton changing the current location to *Activated*. In the latter case, the signal *iota_changed* is issued to synchronise with the *ActivityCalculation*. Finally, there are two outgoing edges of *Activated*—one for a signal via channel *s_changed_internal* and one for a signal via channel *i_changed_internal*. The former case models that B is not stimulated anymore, while the latter models that it is inhibited. Both cases result in B not being activated anymore, which is signalled to the *ActivityCalculation* via channel *iota_changed* on taking the following edge.

While the automata *StimulationInterface*, *InhibitionInterface*, and *ActivationCalculation* model the processing of the incoming behaviour signals s and i , the remaining two automata are dedicated to calculating the outgoing behaviour signals a and r . The one modelling the calculation of a (*ActivityCalculation*) can be seen in Fig. 4.10. It consists of three locations (*Inactive*, *Inactive_but_Activated*, and *Active*) as well as a number of connecting edges. In the initial location, the automaton waits for the *ActivationCalculation* to signal a change of the behaviour’s activation

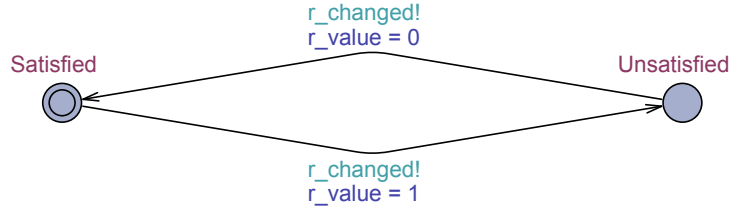


Figure 4.11: The TargetRatingCalculation of a standard behaviour, calculating its target rating.

via channel `iota_changed`. If the signal is received, the automaton will transition to `Inactive_but_Activated`. As mentioned in Sec. 2.2.1, $a_B \leq \iota_B$ at all times. As the `ActivityCalculation` shall respect this rule, a change of a_B from 0 to 1 is only possible in a location where ι_B is 1. For the model of a standard behaviour, a very simple calculation of a has been chosen: In case $\iota_B = 1$, the automaton can switch arbitrarily between a location representing $a_B = 0$ (`Inactive_but_Activated`) and one representing $a_B = 1$ (`Active`). Whenever the automation switches between these two locations, the variable storing the value of the activity (`a_value`) is updated and a signal is sent out via channel `a_changed`. If the activation goes down to 0 while the automaton is in `Inactive_but_Activated`, it transitions back to the initial location. If it is in `Active` while receiving the signal that ι_B has changed to 0, it also transitions back to the initial location, but updates `a_value` and sends out a signal via `a_changed`.

The fifth automaton belonging to the model of a standard behaviour (`TargetRatingCalculation`) is also the most simple one (see Fig. 4.11). It does not wait for any incoming signal, but simply models the target rating as a value that changes arbitrarily between 0 and 1. The idea behind this is that in contrast to a behaviour's activity, its target rating does not necessarily depend on its stimulation and inhibition, i.e. there is no rule analogous to $a_B \leq \iota_B$ for the target rating. As a result, the `TargetRatingCalculation` consists of only two locations (`Satisfied` and `Unsatisfied`) as well as two edges. From the initial location (`Satisfied`), the automaton can transition to `Unsatisfied`. When taking the corresponding edge, it sets `r_value` to 1 and sends out a signal via `r_changed`. From `Unsatisfied`, it can go back to `Satisfied` in a similar fashion.

Of course, the models of the calculation of a behaviour's activity and target rating presented above are very simple. But such simple models are sufficient to answer questions like "When can a behaviour get active?". The verification described in the work at hand targets the interaction of behaviours, not their inner calculations. However, it is possible to model the calculation of activity and target rating in a more realistic fashion if this is helpful or necessary for the verification process and if enough information about the calculation functions inside a behaviour is available. In Secs. 4.2.3 and 4.2.4, this is explained for the two coordinating behaviours fusion behaviour (see Sec. 2.2.1) and conditional behaviour stimulator (see Sec. 3.1.1), respectively.

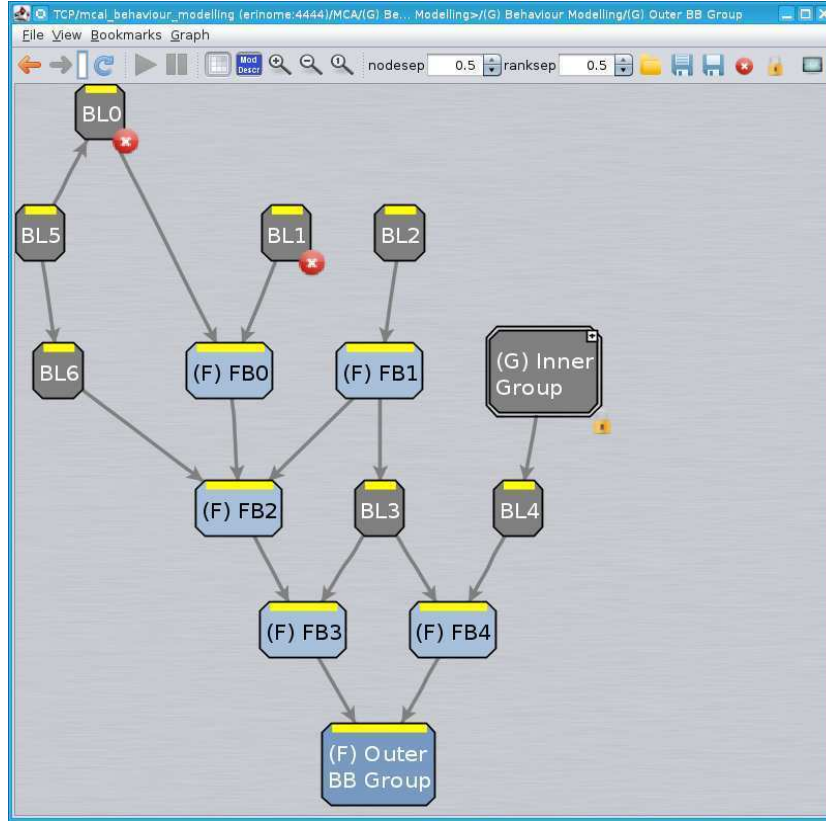


Figure 4.12: The visualisation of a behaviour group in FINSTRUCT. The behaviours that have been marked by the developer with a white cross in a red circle will not be modelled. The behaviour-based group that has been marked with the lock symbol on its lower right corner will be modelled as a standard behaviour, i.e. the behaviours inside this group will not be part of the model.

4.2.2 Mapping iB2C Behaviour Networks to Networks of Uppaal Automata

So far, only the set of automata modelling one standard iB2C behaviour and the corresponding interconnecting channels have been presented. The technical process of creating networks of UPPAAL automata and the technique for connecting automata representing different behaviours have not been described yet. Design Decision 8 requires that the mapping process from an iB2C behaviour network to a network of UPPAAL automata be automatic. Such an automatic mapping process has been developed and implemented (see [Ropertz 11], [Rittmann 12], and [Armbrust 12a]). It is started from a running MCA2-kl or FINROC program that contains the iB2C network to be modelled. Typically, the developer does not want to model the complete behaviour network, but only the part of it that is relevant to the current verification process. Hence, it should be possible to configure the modelling process in an appropriate way.

With regard to the usability of the proposed modelling approach, a concept has been developed for graphically configuring the modelling process. It has been integrated into FINSTRUCT as a special widget (see [Ropertz 12]). Figure 4.12 depicts this widget visualising a behaviour-based group. Using the mouse, the developer can mark behaviours

or behaviour groups with a white cross in a red circle in order to exclude them from the modelling process. This can be done in case the developer does not consider a behaviour as relevant for the verification process. Furthermore, he can mark behaviour groups with a lock symbol, indicating that these groups shall be modelled as standard behaviours, i.e. the behaviours within the groups in question shall not be modelled. In cases where the contents of a group are irrelevant, this option can be used.

Algorithm 4.1: Transferring an iB2C network into a network of UPPAAL automata.

```

input : an iB2C network to model
output: a network of UPPAAL automata corresponding to the input network

1 foreach behaviour  $B$  to be modelled do
2   Check type of  $B$  (standard, fusion behaviour, conditional behaviour stimulator);
3   Check connections of  $B$  (number and type);
4   Determine necessary templates to model  $B$  based on type and connections;
5   foreach UPPAAL template  $T$  necessary to model  $B$  do
6     if  $T$  not created yet then
7       Create  $T$ ;
8       Add  $T$  to model;
9     end
10    Add necessary instantiation(s) of  $T$  to model;
11  end
12 end

```

The core of the modelling process is an algorithm running through the behaviour network in question and processing each encountered behaviour. Algorithm 4.1 depicts the essential steps of this modelling process in an abstract way. For each behaviour B that shall be modelled, the algorithm determines the behaviour's type as well as the number and types of its connections (see Lines 2 and 3). Based on this information, it decides which templates are needed to model B (see Line 4). As mentioned in Sec. 4.2.1, some templates cannot be created once and then used for all occurrences of the respective behaviour type, but have to be generated dynamically based on the number of connected behaviours. An example of this is the **InhibitionInterface**. The number of its locations as well as the number of its edges depends on the number of inhibiting behaviours connected to the behaviour in question. In Line 4 of Alg. 4.1, it is checked whether a dynamically created UPPAAL template is needed for modelling a behaviour. If this is the case, it is checked whether the template in question has already been generated (see Line 6). If necessary, it is created and added to the model (see Lines 7 and 8). For example, when the algorithm encounters a behaviour with two connected inhibiting behaviours for the first time while running through the network, it will create the version of the **InhibitionInterface** depicted in Fig. 4.8. When it encounters another behaviour to which two inhibiting behaviours are connected, the algorithm does not have to create the necessary template again because the version created previously can be used.

In Line 10 of Alg. 4.1, the synchronisation channels between the UPPAAL automata are established. The UPPAAL templates described in this thesis have channels as parameters. In the instantiation of a template, these parameters are set to the correct values (i.e. channels) in order to realise the synchronisation between the automata. This technique is applied for establishing synchronisations between automata belonging to one behaviour as well as between automata belonging to different behaviours, i.e. whenever a template is

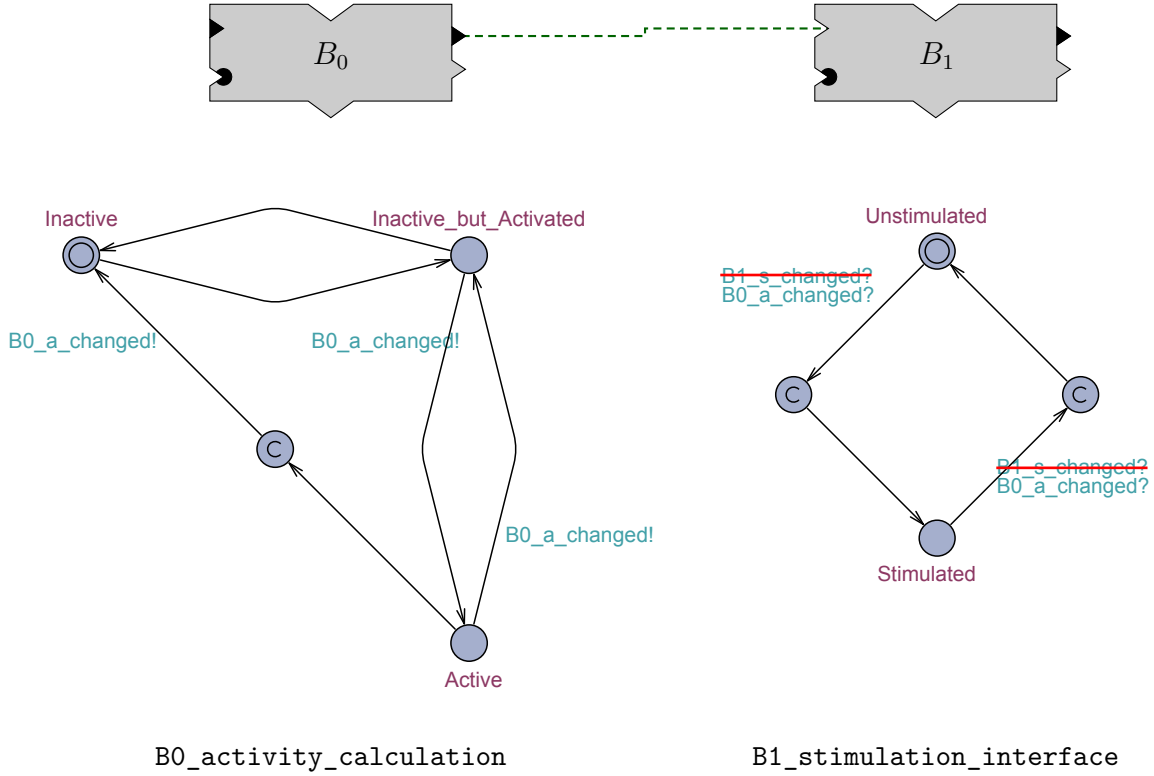


Figure 4.13: B_0 stimulates B_1 with its activity. The automata show how such a connection between two behaviours is modelled. In the `StimulationInterface` of B_1 , `s_changed` is replaced with `a_changed` of B_0 , which is sent out by the `ActivityCalculation` of B_0 .

instantiated for a behaviour B , it is provided with the correct channels depending on the connections of B with other behaviours.

Figure 4.13 illustrates how this works: Behaviour B_0 stimulates behaviour B_1 with its activity, i.e. its activity output is connected to the stimulation input of B_1 . Therefore, there has to be a synchronisation of the instance of `ActivityCalculation` belonging to B_0 and of the instance of `StimulationInterface` belonging to B_1 . When the `ActivityCalculation` of B_0 sends out `a_changed`, indicating that its activity a_{B_0} has changed, the `StimulationInterface` of B_1 has to receive this signal on its channel `s_changed`, indicating that its stimulation s_{B_1} has changed. Therefore, in the instantiation of `StimulationInterface` of B_1 , `s_changed` is replaced with `a_changed` of the `ActivityCalculation` of B_0 . All connections between behaviours are modelled in this way, resulting in a network of UPPAAL automata that are synchronised using channels—in compliance with Design Decision 7. This network is saved to a file that can later be processed by UPPAAL. As UPPAAL uses an XML-based file format (see Sec. 4.1.2.2), creating this file is straightforward.

4.2.3 Mapping iB2C Fusion Behaviours to Uppaal Automata

For the model of a fusion behaviour, three of the automata used in the model of a standard behaviour are reused: the `StimulationInterface`, the `InhibitionInterface`, and the `ActivationCalculation`. Fusion behaviours are important coordination behaviours that

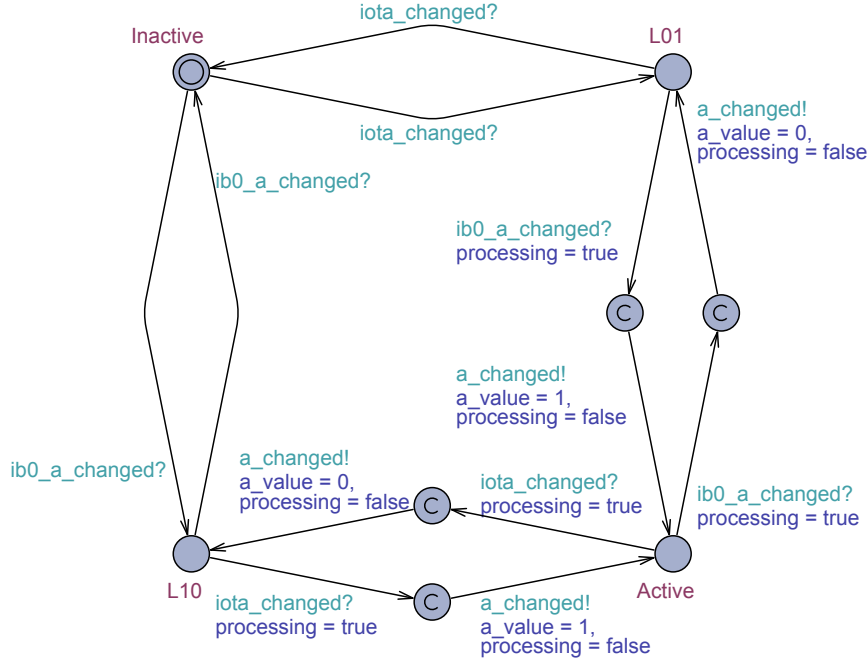


Figure 4.14: The `FBActivityCalculation (Version 1)` of a fusion behaviour with $n_c = 1$ connected input behaviour.

occur frequently in behaviour networks. Therefore, it seems beneficial to model the calculation of their activity and target rating in a more sophisticated way than those of standard behaviours. As already mentioned in Sec. 4.2.1, the proposed modelling approach allows for creating more realistic models of special behaviours. In this section, it is presented how such models can be designed by using special automata for the calculations of a fusion behaviour’s activity and target rating.

In Sec. 2.2.1, it has been explained that iB2C fusion behaviours offer three different fusion methods—maximum, weighted average, and weighted sum. Due to the reduction of the value range of iB2C signals to $\{0, 1\}$ for the modelling and due to often having queries like “Can a behaviour get active at all?”, there is no use in distinguishing between the three methods; instead, all fusion behaviours are modelled in the same way, no matter what their fusion method is. Two approaches for modelling the calculation of the activity of a fusion behaviour have been developed. They differ significantly in the numbers of locations and edges (see Sec. 4.2.5). A detailed description of them will be provided next, followed by a presentation of the automaton modelling the target rating calculation of a fusion behaviour.

4.2.3.1 Modelling the Activity Calculation of a Fusion Behaviour With a Single Automaton

Figure 4.14 depicts a first version of a template modelling the activity of a fusion behaviour to which one input behaviour is connected. It is referred to as the `FBActivityCalculation (Version 1)`. While there is usually no sense in connecting only one input behaviour to a fusion behaviour, this simple example will help in explaining the basic ideas behind the `FBActivityCalculation (Version 1)`.

According to the way fusion behaviours are modelled in UPPAAL, a fusion behaviour B_{Fusion} with n_c connected input behaviours B_{Input_d} ($d \in \{0, \dots, n_c - 1\}$) is active if and only if the following two requirements are fulfilled:

1. B_{Fusion} is activated, i.e. $\iota_{\text{Fusion}} = 1$.
2. At least one of the connected input behaviours is active, i.e. $\exists d : (0 \leq d \leq n_c - 1) \wedge (a_{\text{Input}_d} = 1)$.

These two requirements are reflected in the **FBActivityCalculation (Version 1)**. A transition from the location representing $a_{\text{Fusion}} = 0$ (**Inactive**) to the one representing $a_{\text{Fusion}} = 1$ (**Active**) is only possible if ι_{Fusion} changes from 0 to 1 and the activity a_{Input_0} of the single connected input behaviour B_{Input_0} also changes from 0 to 1. These changes are signalled using channels **iota_changed** and **ib0_a_changed**, respectively. In case **iota_changed** and **ib0_a_changed** have been received (no matter in which sequence), the **FBActivityCalculation (Version 1)** sends out **a_changed**, sets **a_value** to 1, and enters **Active**. If ι_{Fusion} or a_{Input_0} goes back down to 0 (i.e. **iota_changed** or **ib0_a_changed** is received) while the template is in location **Active**, **a_changed** is sent out and a_{Input_0} is reset to 0. As usual, the **processing** flag is used to allow for easily excluding intermediate locations from the results of queries.

Looking at Fig. 4.15, it can be seen how the structure of the **FBActivityCalculation (Version 1)** looks like if B_{Fusion} combines the outputs of $n_c = 2$ competing behaviours. The square structure of the template for $n_c = 1$ has been extended to a cubic structure in which the template depicted in Fig. 4.14 appears on the left hand side in a slightly modified form. With the above remarks about the template for $n_c = 1$ competing behaviours, the one for $n_c = 2$ is self-explanatory. The only aspect that shall be pointed out here is that the structure of the **FBActivityCalculation (Version 1)** for n_c connected behaviours is an $n_c + 1$ -cube or $n_c + 1$ -dimensional hypercube. This characteristic is important for the calculations that will be presented in Sec. 4.2.5. The creation of different variants of the **FBActivityCalculation (Version 1)** for different numbers n_c of competing behaviours is done automatically during the execution of Alg. 4.1 (see Line 7) in the same way as the creation of different variants of the **InhibitionInterface** is done.

4.2.3.2 Modelling the Activity Calculation of a Fusion Behaviour With Multiple Automata

After an approach to modelling the activity of an iB2C fusion behaviour B_{Fusion} with a single UPPAAL automaton has been presented in the previous section, an approach using multiple automata shall be presented in the following (see also [Rittmann 12]). The central idea is to have a separate automaton (**FBIBActivityCalculation**) for each of the n_c connected behaviours B_{Input_d} . Each of these automata checks whether B_{Fusion} is activated and B_{Input_d} is active. If both is the case, a signal is sent to another automaton (**FBActivityCalculation (Version 2)**) that combines the results of the n_c single automata.

Figure 4.16 depicts the **FBIBActivityCalculation**. It is instantiated once for each input behaviour B_{Input_d} connected to a fusion behaviour B_{Fusion} . In its initial state (**Inactive**), it waits for ι_{Fusion} or a_{Input_d} to change from 0 to 1. This is signalled via

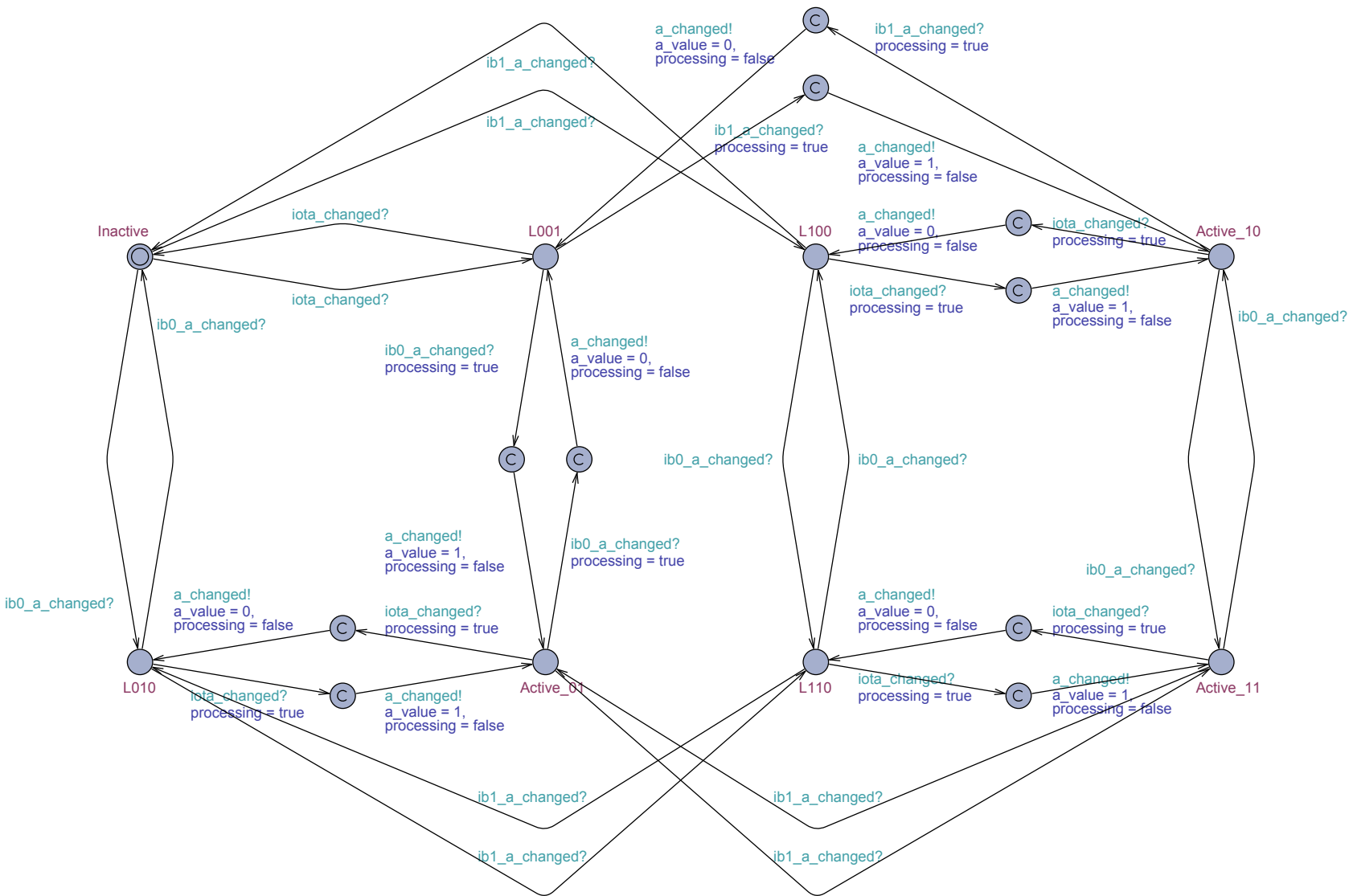


Figure 4.15: The FBActivityCalculation (Version 1) of a fusion behaviour with $n_c = 2$ connected input behaviours.

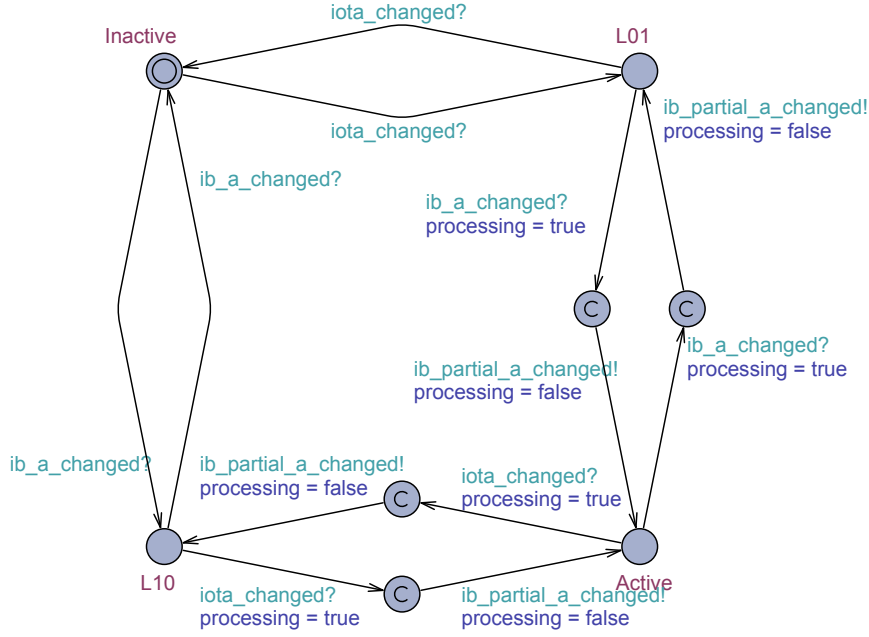


Figure 4.16: The FBIBActivityCalculation of a fusion behaviour B_{Fusion} . It combines the activation ι_{Fusion} of B_{Fusion} with the activity a_{Input_d} of an input behaviour B_{Input_d} .

channels `iota_changed` or `ib_a_changed`, respectively, and causes a transition to L01 or L10, respectively. If the respective other value also changes from 0 to 1, the automaton sends out `ib_partial_a_changed` to synchronise with the FBActivityCalculation (Version 2) and transitions to Active. This location represents that B_{Fusion} is activated and B_{Input_d} is active. The return path is realised in the same way as in the FBActivityCalculation (Version 1) for $n_c = 1$ (see Fig. 4.14). Actually, the two templates are altogether very similar.

The template combining the signals sent by $n_c = 2$ single instances of FBIBActivityCalculation, denoted by FBActivityCalculation (Version 2), is depicted in Fig. 4.17. Starting in the initial location (Inactive), it waits for one of the n_c instances of FBIBActivityCalculation to signal that the corresponding input behaviour is active and that B_{Fusion} is activated. In the case of $n_c = 2$, the signalling is done via channels `ib_0_partial_a_changed` and `ib_1_partial_a_changed`, respectively. If one of the signals is received, the FBActivityCalculation (Version 2) sets the variable modelling the activity of the fusion behaviour (`fb_a_value`) to 1, informs other automata about a change of the activity via `fb_a_changed`, and transitions to either Active_01 or Active_10. A signal from the respective other instance of the FBIBActivityCalculation does not require updating a variable or sending out another signal; it only initiates a transition to Active_11. The way back to the initial location is realised in the usual way. As a matter of course, the FBActivityCalculation (Version 2) has to be created specifically for each number of behaviours connected to a fusion behaviour. The different variants are generated automatically by Alg. 4.1 (see Line 7) in the same way as other templates that have to be created dynamically.

The two variants of modelling the activity calculation of a fusion behaviour presented in the preceding and this section—using a single automaton or using multiple automata—differ

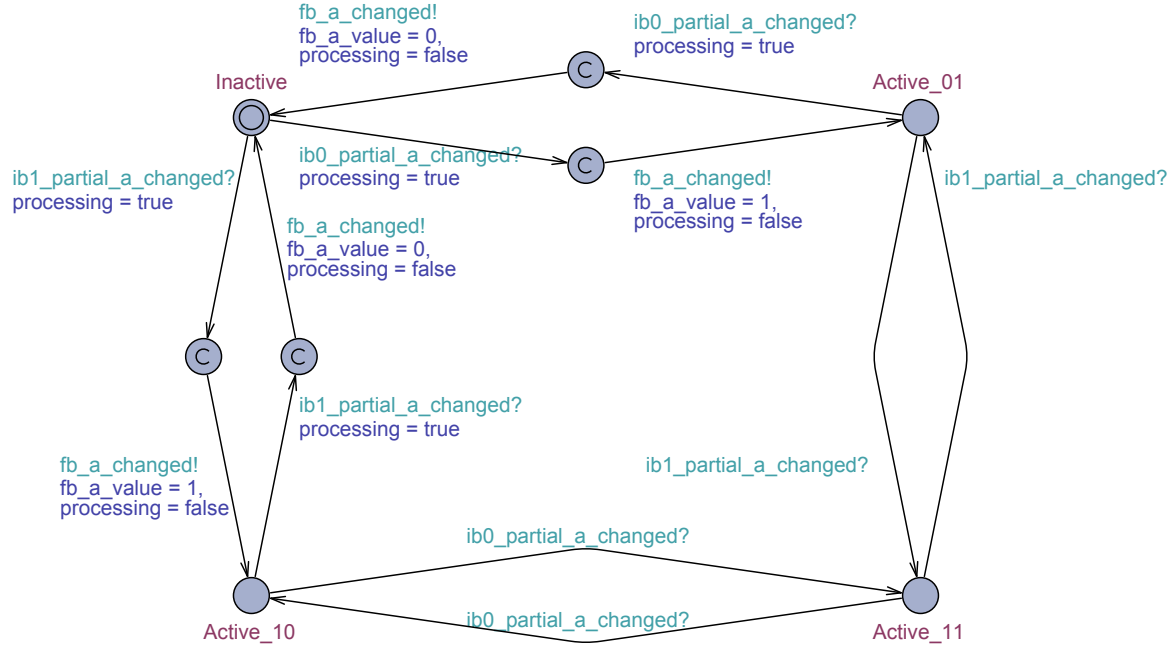


Figure 4.17: The FBActivityCalculation (Version 2) of a fusion behaviour B_{Fusion} with $n_c = 2$ connected behaviours. It combines the outputs of the single instances of the FBIBActivityCalculation, takes care of setting `fb_a_value` to the correct value, and synchronises with other automata via `fb_a_changed` whenever `fb_a_value` has changed.

in the numbers of locations and edges needed for a given number of input behaviours that are connected to a fusion behaviour. Section 4.2.5 provides information about the complexity of the involved automata and about when one or the other model should be used.

4.2.3.3 Modelling the Target Rating Calculation of a Fusion Behaviour With a Single Automaton

While its dependence on the activation suggests a distribution of the activity calculation over different automata as described in the previous section, there is no such dependence for the calculation of the target rating. Hence, only one way of modelling it is described here.

According to Eq. 2.3 on P. 27, the target rating of a fusion behaviour realising a maximum fusion is set to the target rating of the connected input behaviour with the highest activity. As a detailed analysis of the target rating using the verification approach presented in this work had not been considered necessary, a model has been chosen that resembles the one used for calculating the inhibition of a behaviour (see Fig. 4.8). Therefore, it also has the basic structure of a hypercube. The resulting template, `FBTargetRatingCalculation`, is based on the fact that a fusion behaviour can only be unsatisfied if at least one of its input behaviours is unsatisfied. In Fig. 4.18, the version for a fusion behaviour B_{Fusion} with $n_c = 2$ connected behaviours is depicted.

The initial location symbolises that the fusion behaviour is satisfied, which here corresponds to all input behaviours being satisfied. If a connected behaviour signals a change of its

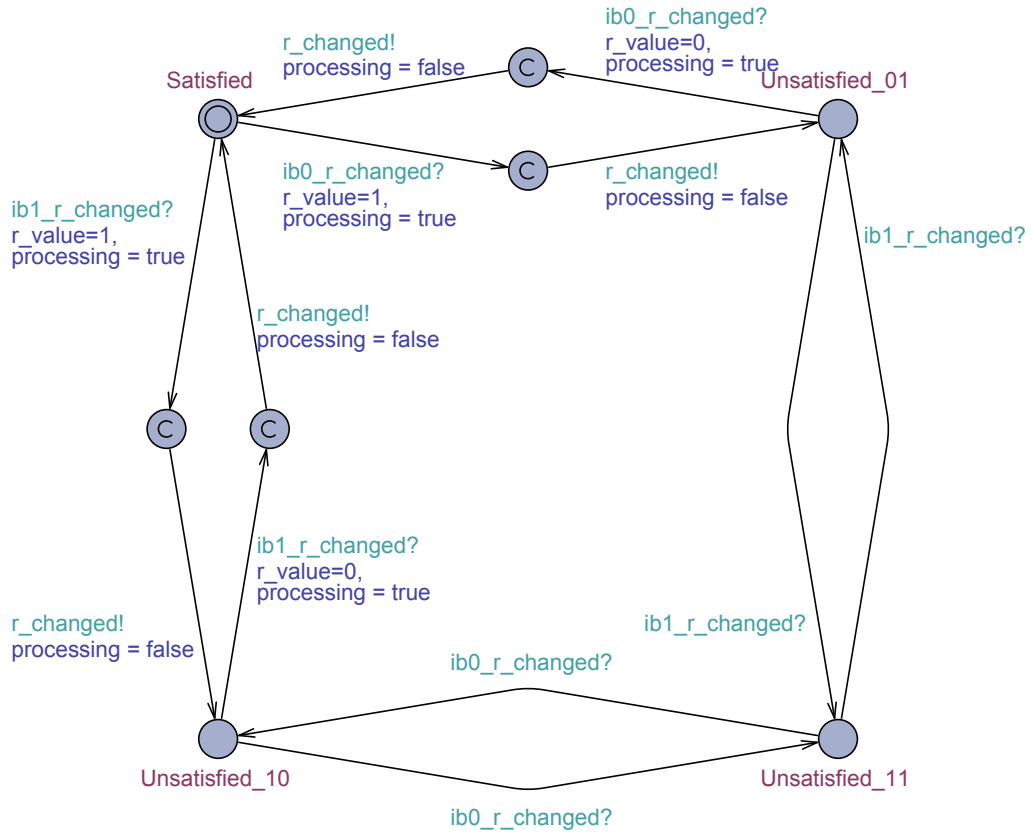


Figure 4.18: The FBTargetRatingCalculation of a fusion behaviour B_{Fusion} with $n_c = 2$ connected behaviours.

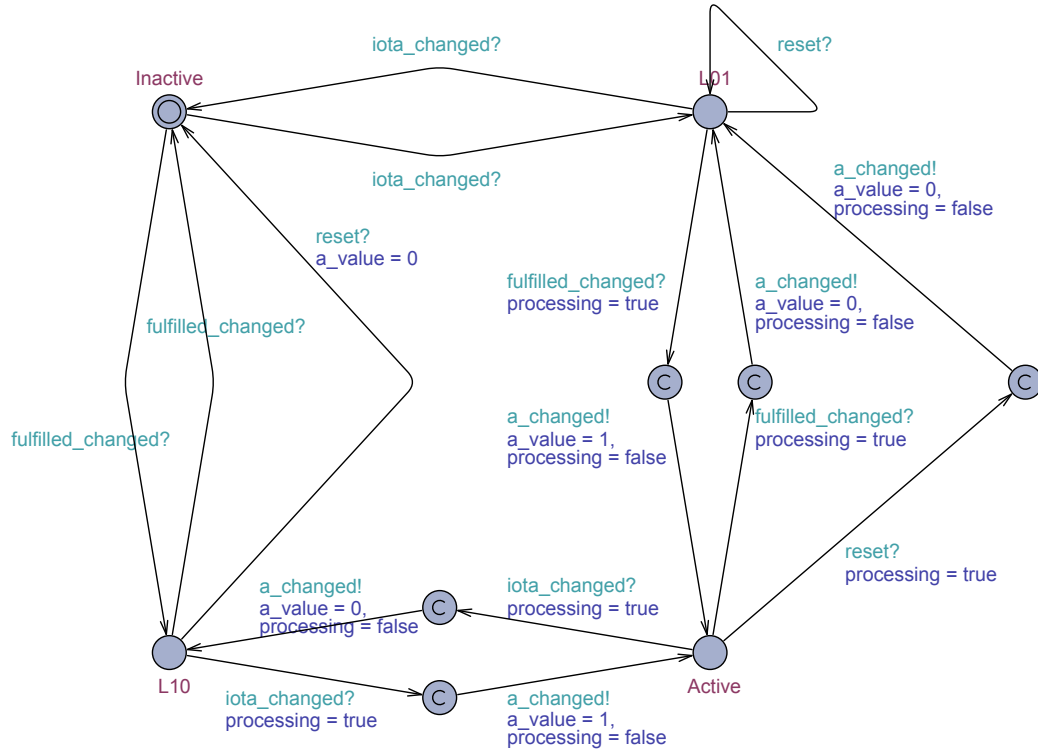


Figure 4.19: The CBSActivityCalculation of a CBS behaviour.

target rating from 0 to 1, the `FBTargetRatingCalculation` updates `r_value` and informs other automata via `r_changed`. Depending on whether the first or the second input behaviour changed its target rating, the template transitions to `Unsatisfied_01` or `Unsatisfied_10`. The template's behaviour in case of a change of the target rating of the respective other behaviour is intuitive: It simply transitions to `Unsatisfied_11`. The way back to `Unsatisfied_01`, `Unsatisfied_10`, or `Satisfied` is also realised in a way similar to templates already described above.

4.2.4 Mapping iB2C CBS Behaviours to Uppaal Automata

In this section, the modelling of the conditional behaviour stimulator (CBS) introduced in Sec. 3.1.1 is described. As has been explained in said section, the ports of the CBS have changed slightly since its first description in [Armbrust 11a]. Its UPPAAL counterpart models the reset input, but also features permanent feedback conditions in order to support legacy iB2C behaviour networks.

The CBS is much more complex than the fusion behaviour. A significant part of its complexity is caused by the fact that it has ports for six different types of conditions. To each of them, an arbitrary number of behaviours can be connected. Due to this complexity, the model of the CBS consists of more automata than the model of the fusion behaviour.

For the sake of clarity, the following explanations will assume that to each port associated with enabling, ordering, or permanent input or feedback conditions exactly one behaviour is connected. How the model of the CBS can be adapted so that an arbitrary number of behaviours can be connected is explained further below.

The templates modelling stimulation, inhibition, and activation of a CBS node B_{CBS} are the same as for a standard or a fusion behaviour: **StimulationInterface**, **InhibitionInterface**, and **ActivationCalculation**. However, the template modelling the activity, **CBSActivityCalculation**, differs. It is depicted in Fig. 4.19. As can be seen, its structure is similar to those of other templates described before. Again, the initial location (**Inactive**) represents that B_{CBS} is inactive, i.e. that $a_{\text{CBS}} = 0$. A transition to one of two locations can be triggered by the reception of one of two signals: **iota_changed** and **fulfilled_changed**. While the former also exists in the models of standard and fusion behaviours, the latter is a signal that only exists in the model of the CBS. It is issued by another automaton of the CBS model, **CBSConditionsFulfilled** (see Fig. 4.25 and the corresponding explanations below), and indicates that the status of the input conditions (“all fulfilled” or “not all fulfilled”) has changed. If $\iota_{\text{CBS}} = 1$ and all input conditions are fulfilled, the automaton sets **a_value** to 1, issues **a_changed**, and transitions to **active**. The way back is realised in the usual way. What should also be mentioned here are the edges labelled with **reset?**: As the name of the channel suggests, it is used to reset the automaton to a location indicating that the input conditions are not all fulfilled. It can be triggered by an external automaton, but is in particular sent out by **CBSConnectICAndFC** (see below and Fig 4.26), another automaton belonging to the model of the CBS.

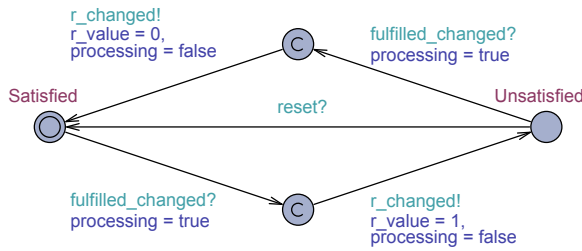


Figure 4.20: The **CBSTargetRatingCalculation** of a CBS behaviour.

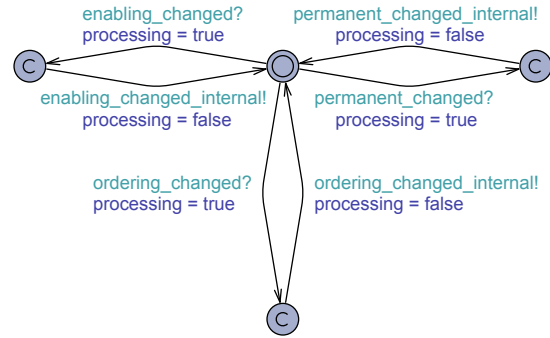


Figure 4.21: The **CBSInputChangedInterface** of a CBS behaviour. It translates external signals to signals used for synchronising the automata belonging to the model of a CBS. In the depicted case, there are three connected behaviours—one for each type of condition (enabling, ordering, and permanent).

The automaton modelling the calculation of the target rating of a CBS—**CBSTargetRatingCalculation**—is simple (see Fig. 4.20): Starting from the initial location (**Satisfied**), the automaton transitions to **Unsatisfied** upon reception of **fulfilled_changed**, sets **r_value** to 1 and sends out **r_changed**. If **fulfilled_changed** is received again, the automaton resets **r_value** to 0, sends out **r_changed**, and returns to the initial state.

Figure 4.21 shows another simple automaton belonging to the UPPAAL model of a CBS: the **CBSInputChangedInterface**. Its purpose is to translate external signals (i.e. signals from automata not belonging to the model of the CBS in question) to internal signals (i.e. signals used for synchronising automata belonging to the model

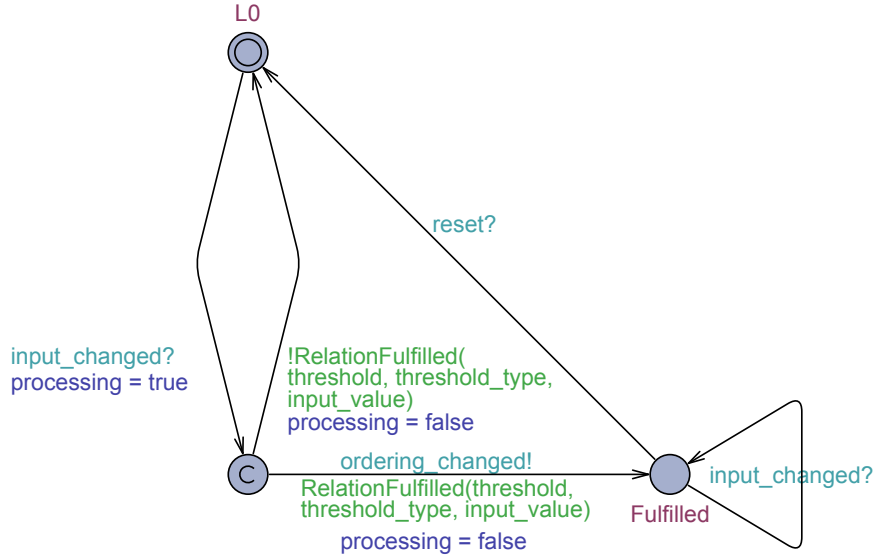


Figure 4.22: CBSOrderingFulfilled of a CBS behaviour. It takes care of checking whether the associated ordering condition is fulfilled or not.

of the CBS in question). It does this by waiting for a signal that the input associated with an enabling, ordering, or permanent condition has changed, signalled via `enabling_changed`, `ordering_changed`, or `permanent_changed`, respectively. This signal is sent out by the model of a behaviour connected to the CBS and *not* by one of the automata `CBSEnablingFulfilled`, `CBSOrderingFulfilled`, and `CBSPermanentFulfilled` (see below), which use *different* signals of the same names. Upon reception of such an external signal, the `CBSInputChangedInterface` sends out the corresponding internal signal as a notification for other automata of the same CBS. `CBSInputChangedInterface` is instantiated twice for each CBS: once for the input conditions and once for the feedback conditions. The advantage of this translation of signals is a better encapsulation: For the synchronisation between the automata belonging to one CBS, only internal signals are used. This facilitates the automatic connection of automata that is performed in Line 10 of Alg. 4.1.

The following three templates wait for a change of the input value of the associated behaviour, upon which they check whether the corresponding condition is fulfilled. As there are three different types of conditions (enabling, ordering, or permanent), three different templates are needed. The simplest one is `CBSOrderingFulfilled` (see Fig. 4.22). For ordering conditions, a fulfilment of the relation causes the condition to also be fulfilled. If then the relation is not fulfilled anymore, this does not change the fulfilment of the condition. When the automaton is in the initial location and the input value changes (indicated via `input_changed`), the automaton checks whether the corresponding relation is fulfilled using a call to `RelationFulfilled`. If it is *not* fulfilled, the automaton stays in the initial location. If it is fulfilled, the automaton sends out `ordering_changed` and transitions to location `Fulfilled`, where it stays until a reset signal is received. The template is instantiated once for the ordering input condition and once for the ordering feedback condition.

The template checking a permanent condition (`CBSPermanentFulfilled`) is realised in a

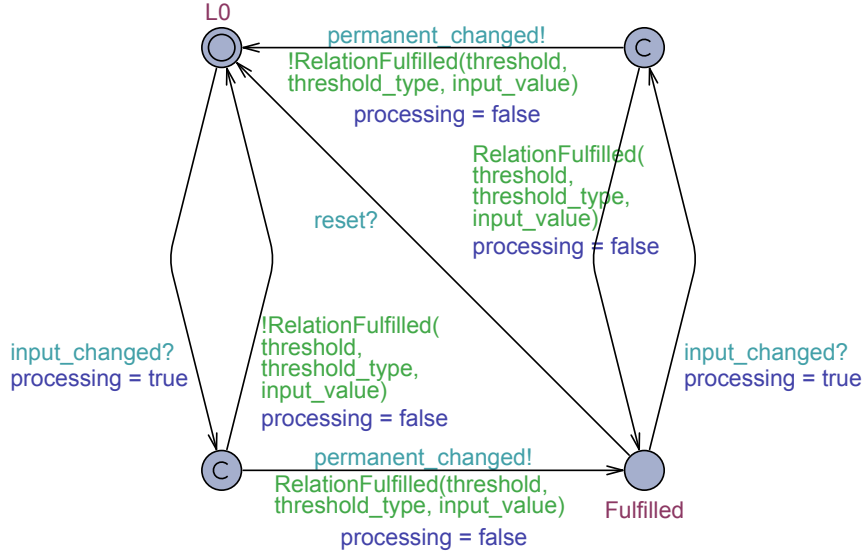


Figure 4.23: CBSPermanentFulfilled of a CBS behaviour. It takes care of checking whether the associated permanent condition is fulfilled or not.

similar way as CBSOrderingFulfilled. In contrast to an ordering condition, a permanent condition does not stay fulfilled in case the corresponding relation is not fulfilled anymore. This has to be reflected in the template. As can be seen in Fig. 4.23, the structure of the template is partly the same as the one of CBSOrderingFulfilled. The only structural difference is that in location Fulfilled, a change of the input value triggers a call to RelationFulfilled for checking whether the relation is still fulfilled. Depending on the result of the check, the automaton stays in Fulfilled or issues permanent_changed and returns to the initial location. This template is also instantiated once for the permanent input condition and once for the permanent feedback condition.

The most complex of the three templates, CBSEnablingFulfilled, is responsible for checking the fulfilment of an enabling condition. Its complexity arises from the fact that in contrast to the fulfilment of an ordering or a permanent condition, the fulfilment of an enabling condition does not only depend on the corresponding relation, but also on the fulfilment of the ordering and permanent conditions. As can be seen in Fig. 4.24, there are not only edges for synchronising via input_changed, but also ones for synchronising via ordering_changed or permanent_changed. The latter two signals are sent out by CBSOrderingFulfilled and CBSPermanentFulfilled, respectively (see above). As soon as the ordering as well as the permanent condition and the enabling relation are fulfilled, CBSEnablingFulfilled sends out enabling_changed. The template also contains several edges leading back to the initial location in case reset is received. Just as the other two templates for checking the fulfilment of conditions, CBSEnablingFulfilled is instantiated once for the enabling input condition and once for the enabling feedback condition.

The signals enabling_changed, ordering_changed, and permanent_changed are combined by an instance of CBSConditionsFulfilled, which is shown in Fig. 4.25. As soon as CBSConditionsFulfilled has received the signals indicating that all conditions are fulfilled, it issues fulfilled_changed, the signal that is essential for the CBSActivityCalculation (see above). The structure of the template contains no surprises;

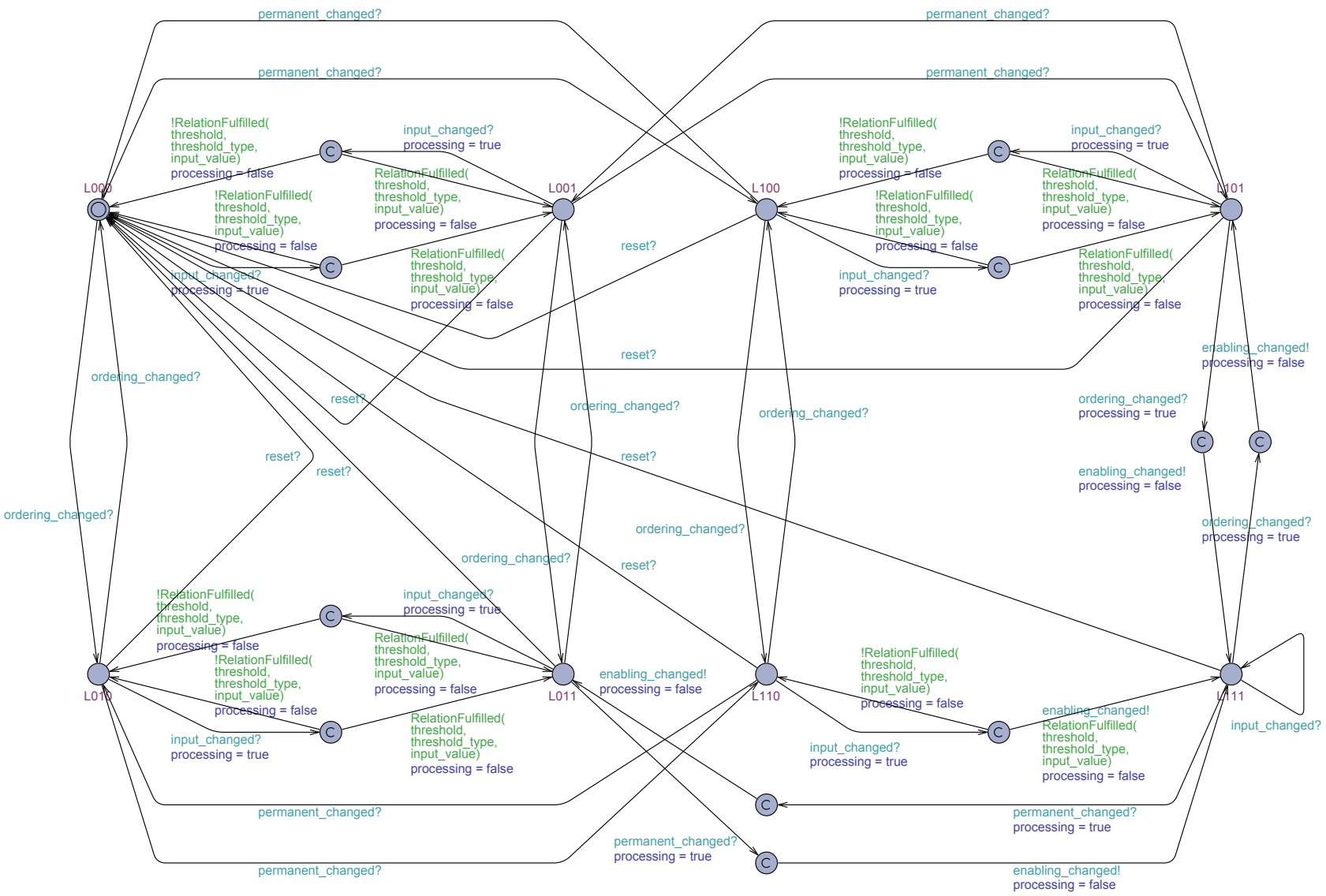


Figure 4.24: CBSEnablingFulfilled of a CBS behaviour. It takes care of checking whether the associated enabling condition is fulfilled or not.

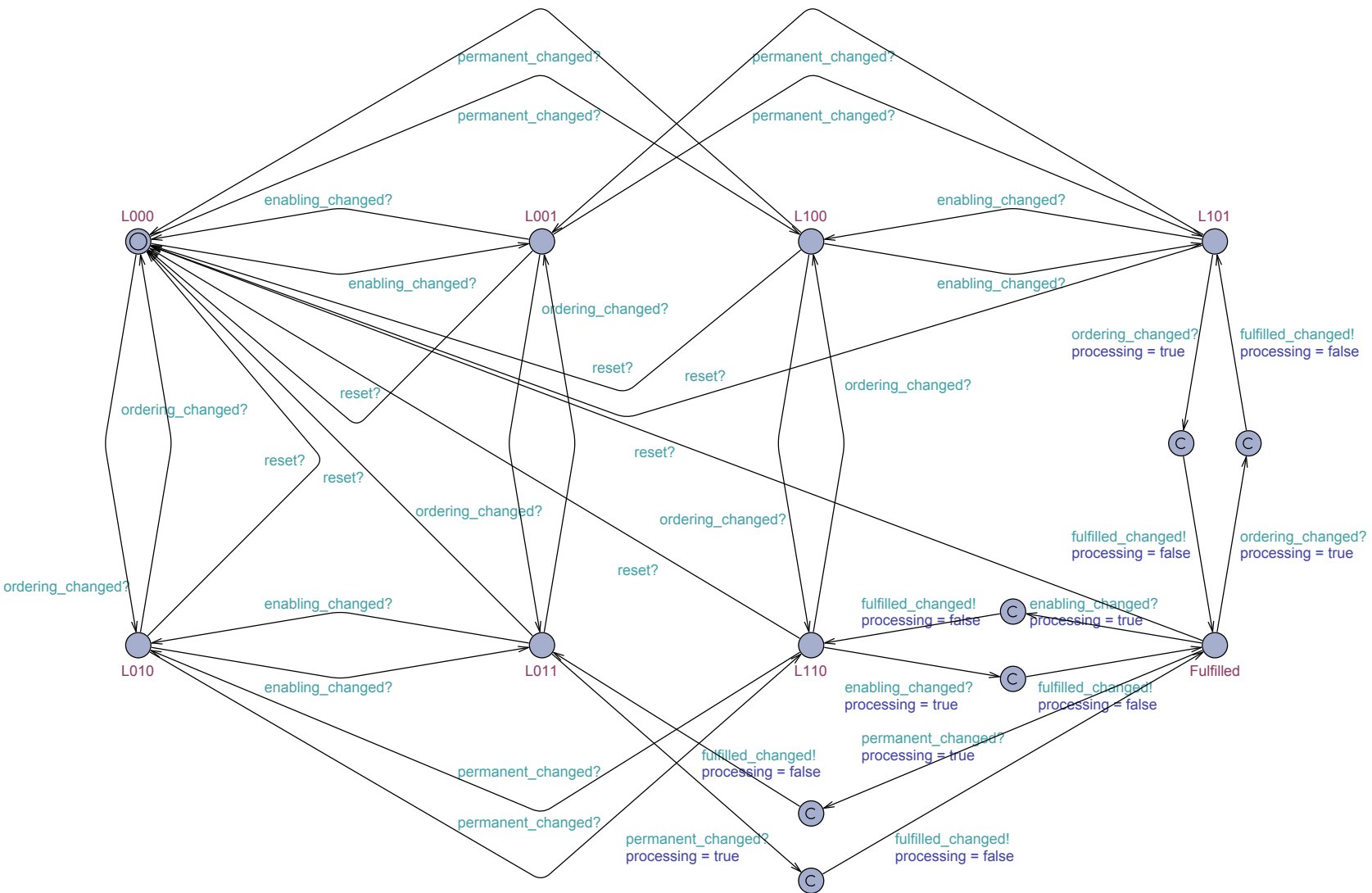


Figure 4.25: CBSConditionsFulfilled of a CBS behaviour. It combines the signals sent by CBSEnablingFulfilled, CBSOrderingFulfilled, and CBSPermanentFulfilled.

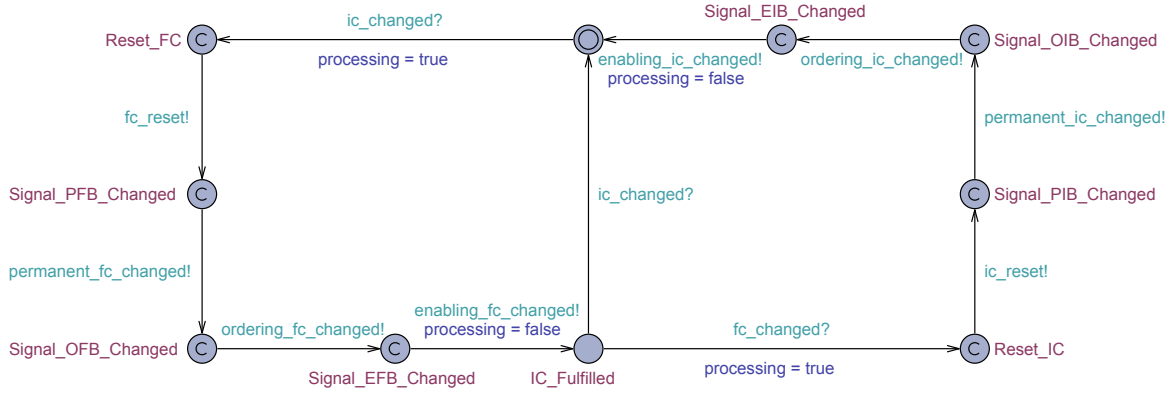


Figure 4.26: CBSCoordinateICAndFC of a CBS behaviour. It synchronises the two instances of CBSConditionsFulfilled dealing with the fulfilment of the input conditions and the feedback conditions, respectively.

as a lot of functionality has been sourced out to CBSEnablingFulfilled, CBSOrderingFulfilled, and CBSPermanentFulfilled, it only has a moderate size and complexity. CBSConditionsFulfilled is instantiated twice: once for the input conditions and once for the feedback conditions.

For reasons of clarity, many of the automata modelling a CBS node are instantiated once for the input conditions and once for the feedback conditions. Each of the two instances of CBSConditionsFulfilled combines the signals sent by the automata belonging to its part. But what is missing so far is an automaton that synchronises these two instances or, in other words, that connects the two parts of the model. This automaton is called CBSCoordinateICAndFC and is depicted in Fig. 4.26. Starting in the initial location, it waits for the signal `ic_changed`, which is corresponding to `fulfilled_changed` of the instance of CBSConditionsFulfilled that deals with the input conditions. The association between the two signals is established in Line 10 of Alg. 4.1 by passing the same channel as argument to the two templates during the instantiation. A change of the fulfilment of the input conditions in the initial location corresponds to all input conditions of the CBS node becoming fulfilled. If this happens, the CBS node starts checking the feedback conditions. Therefore, the automata modelling the feedback conditions are reset to their initial states by sending `fc_reset` and a check of the conditions is triggered by sending out `permanent_fc_changed`, `ordering_fc_changed`, and `enabling_fc_changed`, which are associated with the `input_changed` channels of those instances of CBSPermanentFulfilled, CBSOrderingFulfilled, and CBSEnablingFulfilled, respectively, that deal with feedback conditions. This is necessary as these automata have to re-evaluate the fulfilment of their conditions after they have been reset. After this has been done, CBSCoordinateICAndFC will be in the location `IC_Fulfilled`. If it then receives the signal `ic_changed` again (indicating that not all input conditions are fulfilled anymore), it will go back to the initial location. However, if it receives the signal `fc_changed`, indicating that the feedback conditions are fulfilled, it will reset the automata dealing with the fulfilment of the input conditions and thereupon enter the initial location.

So far, it has been assumed that to each port of a CBS associated with enabling, ordering, or permanent input or feedback conditions exactly one behaviour is connected. In order

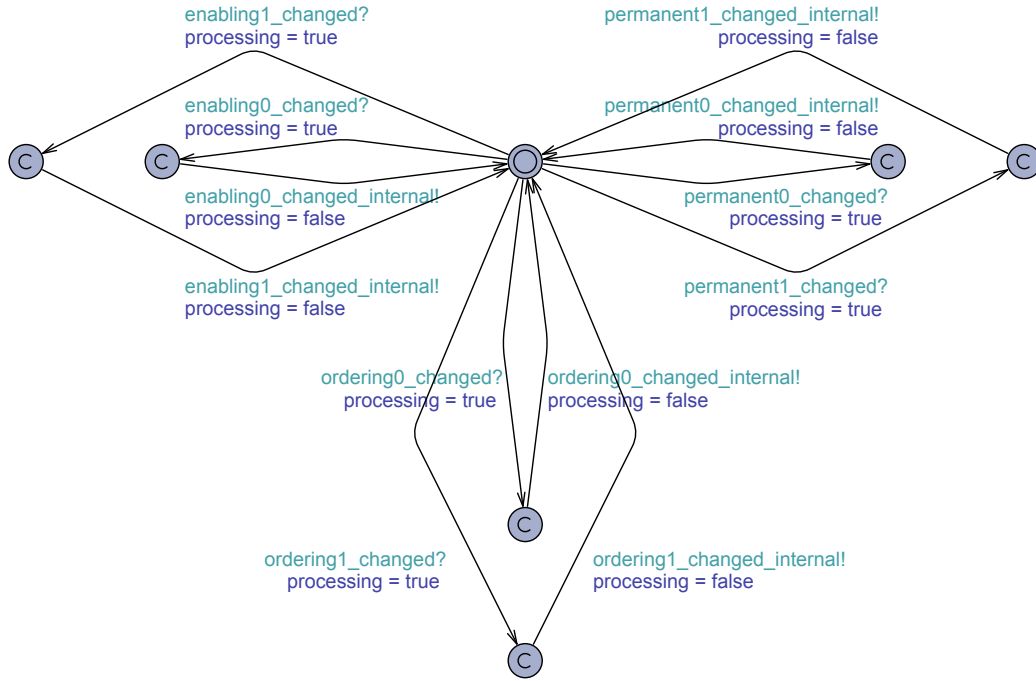
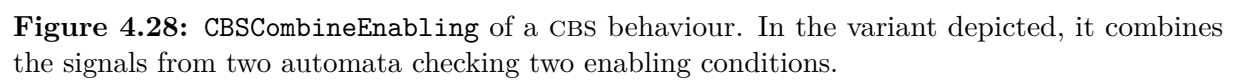


Figure 4.27: The `CBSInputChangedInterface` of a CBS behaviour. It translates external signals to signals used for synchronising the automata belonging to the model of a CBS. In this case, there are six connected behaviours—two for each type of condition (enabling, ordering, and permanent).

to support the connection of an arbitrary number of behaviours, the model of the CBS has to be adapted. How this can be done will be explained in the following (see also [Rittmann 12]). Not all of the automata modelling a CBS have to be altered in order to support the connection of an arbitrary number of behaviours. For example, the general templates `StimulationInterface`, `InhibitionInterface`, and `ActivationCalculation` can remain unchanged. Furthermore, the CBS-specific templates `CBSActivityCalculation` and `CBSTargetRatingCalculation` do not have to be altered.

By contrast, the `CBSInputChangedInterface` has to be adapted. It is created automatically depending on the number of connected behaviours. Figure 4.27 shows the version needed for two enabling, two ordering, and two permanent conditions. As can be seen, the structure resembles the version depicted in Fig. 4.21; another committed location along with two edges connecting it to the initial location has been added for each additional condition. In the same way, a version of the `CBSInputChangedInterface` can be created for each combination of conditions.

The next step is to adapt the automata which check the fulfilment of the different conditions (`CBSEnablingFulfilled`, `CBSOrderingFulfilled`, and `CBSPermanentFulfilled`). One way to do this would be to extend the three automata and create a special version for each number of conditions of the corresponding type. However, this would lead to very complicated automata very fast—especially in the case of `CBSEnablingFulfilled`. Therefore, another approach is followed here: The automata are not extended. Instead, for each condition an instance of the appropriate automaton is created and the fulfilment of the associated condition is signalled by the automaton just like in the simple case described



above. The difference now is that the signal is not directly processed by an instance of `CBSConditionsFulfilled`, but by an automaton that combines all signals belonging to one type of condition. There is one such combining automaton for each type of condition (enabling, ordering, and permanent input and feedback), i.e. in total, there are six such automata. Figure 4.28 exemplarily shows `CBSCombineEnabling`, the automaton combining the signals issued by the instances of `CBSEnablingFulfilled`. The depicted variant combines the signals of two instances of `CBSEnablingFulfilled`. As can be seen, the template is created in a straightforward way. Starting in the initial location, it waits for either of the two connected instances of `CBSEnablingFulfilled` to signal the fulfilment of the corresponding condition. If the automaton then receives the signal that the other condition is also fulfilled, it issues `enabling_changed` and transitions to `Fulfilled`. The way back to the initial location is realised in the usual way. Furthermore, the automaton can be reset using the `reset` signal. The structure of `CBSCombineOrdering` and `CBSCombinePermanent` is exactly the same. The signal indicating whether all conditions of the respective type are fulfilled (`enabling_changed` in the case of `CBSCombineEnabling`) is sent to the corresponding instance of `CBSConditionsFulfilled`. Hence, `CBSConditionsFulfilled` does not have to be updated. As the fulfilment of enabling conditions—in contrast to the fulfilment of ordering or permanent conditions—not only depends on the fulfilment of the associated relations, but also on the fulfilment of other conditions, three different variants of `CBSEnablingFulfilled` are needed now: one for the case that there are only enabling conditions, one for the case that there are only enabling and either ordering or permanent conditions, and one for the case that there are all three types of conditions. The different variants are all static (i.e. do not depend on the number of conditions). Hence, the two additional variants can be created once and then reused whenever needed.

Finally, another template has to be adapted in order to cope with an arbitrary number of conditions: `CBSConnectICAndFC`. Instead of sending signals to exactly six automata so that they re-evaluate their associated conditions, the number of these automata is now arbitrary. Therefore, a new version of the template has to be created automatically for each combination of conditions. Just like in the old version, the sending of signals to the automata checking the fulfilment of conditions is realised as a succession of edges with associated signals. As the structure can be easily derived from the one shown in Fig. 4.26, no other version is depicted here.

4.2.5 Quantitative Aspects

A big challenge for verification approaches based on model checking is the state explosion problem. Therefore, a relevant question is how large (in terms of locations) the UPPAAL model of a given iB2C network gets. The complexity of an automaton does not only depend on the number of its locations, but also on the amount of edges. An automaton with more edges can be more difficult to assess and can thus complicate the verification process. For this reason, the number of locations and the number of edges are important measures when estimating the size of a model or when comparing different models. Hence, quantitative aspects of the modelling concept described above shall be presented next (see also [Rittmann 12] and [Armbrust 13a]). The numbers given represent the worst case and do not take into account technical improvements for special cases (see comment further below). They are summarised in Table 4.1.

The numbers of locations and edges of the **StimulationInterface** can be directly seen from Fig. 4.7: There are 4 locations and also 4 edges. Likewise, the size of the **ActivationCalculation** can be determined from Fig. 4.9: It consists of 4 locations and 7 edges. The **ActivityCalculation** and the **TargetRatingCalculation** of a standard behaviour are also static templates. Looking at Figs. 4.10 and 4.11 yields 4 locations and 6 edges for the **ActivityCalculation** as well as 2 locations and 2 edges for the **TargetRatingCalculation**.

For the **InhibitionInterface**, some calculations are necessary as this template is created dynamically depending on the number n_i of inhibitory behaviours connected to a behaviour B . As mentioned in Sec. 4.2.1, the **InhibitionInterface** has the structure of a hypercube of dimension n_i . A hypercube of dimension n_i has 2^{n_i} vertices and $n_i \cdot 2^{n_i-1}$ edges. The edges in the **InhibitionInterface** are bidirectional, therefore the number of edges has to be multiplied by 2. Furthermore, a committed location and an additional edge are needed whenever i_B changes, i.e. for the cases when all inhibitory behaviours are inactive *and* one gets active or when only one inhibitory behaviour is active *and* this behaviour gets inactive. Hence, for each additional inhibiting behaviour (i.e. for each increase of n_i), two further committed locations with two further edges have to be added. In total, this yields $2^{n_i} + 2 \cdot n_i$ locations and $n_i \cdot 2^{(n_i-1)} \cdot 2 + 2 \cdot n_i = n_i \cdot 2^{n_i} + 2 \cdot n_i = n_i \cdot (2^{n_i} + 2)$ edges.

With these calculations, the numbers of locations and edges for a network of UPPAAL automata modelling an iB2C network that consists of interconnected standard behaviours can be determined. For the model of the iB2C fusion behaviour some additional calculations are necessary. As has been described in Sec. 4.2.3, there are two ways of modelling the calculation of the activity of a fusion behaviour B_{Fusion} . A closer inspection helps to determine in which cases which method is preferable.

It has already been mentioned in Sec. 4.2.3.1 that the structure of the **FBActivityCalculation (Version 1)** (see Fig. 4.15) for n_c connected behaviours is an $n_c + 1$ -dimensional hypercube. This number results from the n_c connected behaviours and the activation ι_{Fusion} , which also has to be taken into account. Similar to the **InhibitionInterface**, committed locations with connecting edges have to be added to the basic structure. The addition of a location and an edge is necessary in all cases where the activity a_{Fusion} changes. There are two types of such cases:

1. ι_{Fusion} changes from 0 to 1 or vice versa *and* at least one of the connected behaviours is active. $\rightarrow 2 \cdot (2^{n_c} - 1)$ cases
2. The activity of one connected behaviour changes from 0 to 1 or vice versa *and* none of the other behaviours is active *and* $\iota_{\text{Fusion}} = 1$. $\rightarrow 2 \cdot n_c$ cases

Together with the 2^{n_c+1} locations of the basic hypercube structure, this yields $2^{n_c+1} + 2 \cdot (2^{n_c} - 1) + 2 \cdot n_c = 2^{n_c+1} + 2^{n_c+1} - 2 + 2 \cdot n_c = 2^{n_c+2} + 2 \cdot (n_c - 1)$ locations. The basic structure has $(n_c + 1) \cdot 2^{n_c}$ bidirectional edges, i.e. there are $(n_c + 1) \cdot 2^{n_c+1}$ unidirectional edges. Taking into account the two above cases, the number of edges of the **FBActivityCalculation (Version 1)** is $(n_c + 1) \cdot 2^{n_c+1} + 2 \cdot (2^{n_c} - 1) + 2 \cdot n_c = (n_c + 1) \cdot 2^{n_c+1} + 2^{n_c+1} - 2 + 2 \cdot n_c = (n_c + 2) \cdot 2^{n_c+1} + 2 \cdot (n_c - 1)$.

The calculations for the second method of modelling a fusion behaviour (see Sec. 4.2.3.2) are as follows: The **FBIBActivityCalculation** has a fixed number of locations (8) and edges

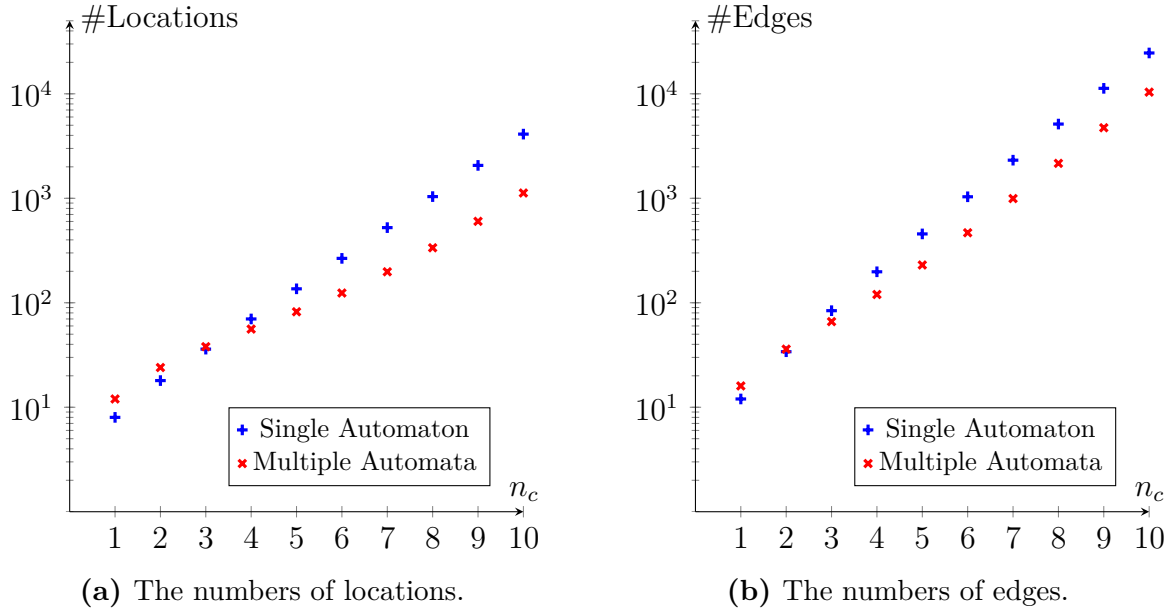


Figure 4.29: The graphs depict the numbers of locations (see Fig. a) and edges (see Fig. b), respectively, needed for modelling the calculation of the activity of an iB2C fusion behaviour with n_c connected behaviours. The blue plus signs correspond to the modelling with one single UPPAAL automaton (`FBAActivityCalculation (Version 1)`), while the red x's correspond to the modelling with multiple automata (n_c instances of `FBIBActivityCalculation` and one instance of `FBAActivityCalculation (Version 2)`).

(12) (see Fig. 4.16). For n_c connected behaviours, it has to be instantiated n_c times, resulting in $8 \cdot n_c$ locations and $12 \cdot n_c$ edges. The `FBAActivityCalculation (Version 2)` for n competing behaviours has the same structure as the `InhibitionInterface` for n inhibitory behaviours. Hence, the `FBAActivityCalculation (Version 2)` has $2^{n_c} + 2 \cdot n_c$ locations and $n_c \cdot (2^{n_c} + 2)$ edges in case of n_c competing behaviours connected to B_{Fusion} . In total, this yields $8 \cdot n_c + 2^{n_c} + 2 \cdot n_c = 2^{n_c} + 10 \cdot n_c$ locations and $12 \cdot n_c + n_c \cdot (2^{n_c} + 2) = (2^{n_c} + 14) \cdot n_c$ edges for the second method of modelling an iB2C fusion behaviour.

The question now is which of the two methods is preferable under which circumstances. Therefore, the numbers of locations and edges of the two have to be compared. As can be seen in Fig. 4.29a, for $n_c \geq 4$, the method using multiple automata to model the activity calculation of a fusion behaviour needs less locations than the method using only a single automaton. Figure 4.29b shows that for $n_c \geq 3$, the method based on multiple automata needs less edges than the one based on a single automaton. The number of locations is crucial with respect to the size of the state space. Therefore, the optimal solution would be to model the activity calculation of fusion behaviours with $n_c < 4$ connected behaviours with one single instance of `FBAActivityCalculation (Version 1)`, while using the method based on n_c instances of `FBIBActivityCalculation` and one instance of `FBAActivityCalculation (Version 2)` for $n_c \geq 4$. In the current implementation, however, the latter method is always used in order to reduce the complexity of the implementation. This decision can be justified by the fact that using the second method for $n_c < 4$ causes only a small overhead compared to the first method, while it heavily reduces the complexity for higher values of n_c .

Finally, the numbers of locations and edges of the `FBTargetRatingCalculation` have to be

determined. Like the `InhibitionInterface` and the `FBActivityCalculation (Version 2)`, the basic structure of the `FBTargetRatingCalculation` is a hypercube—in this case of dimension n_c (cp. Sec. 4.2.3.3). Again, the edges are bidirectional and a committed location along with an edge has to be added in case r_{Fusion} changes. There are $2 \cdot n_c$ such cases. In total, this yields $2^{n_c} + 2 \cdot n_c$ locations and $2 \cdot 2^{n_c-1} \cdot n_c + 2 \cdot n_c = (2^{n_c} + 2) \cdot n_c$ edges.

Of the behaviours described in the work at hand, the conditional behaviour stimulator (see Sec. 3.1.1 for the description of the behaviour and Sec. 4.2.4 for the description of its model) is the one with the most complex UPPAAL model. This is mostly due to the different ways of how behaviours can be connected to it. Therefore, several calculations have to be performed in order to determine the total numbers of locations and edges of the UPPAAL model of a CBS. Like in Sec. 4.2.4, the following calculations refer to the CBS node that featured permanent feedback conditions as described in the original publication (see [Armbrust 11a]). The reason for this is that the modelling algorithm can still process permanent feedback conditions in order to support legacy iB2C systems. In the following, it shall be assumed that a CBS has n_{EIC} enabling, n_{OIC} ordering, and n_{PIC} permanent input conditions as well as n_{EFC} enabling, n_{OFC} ordering, and n_{PFC} permanent feedback conditions. This yields $n_{\text{IC}} = n_{\text{EIC}} + n_{\text{OIC}} + n_{\text{PIC}}$ input conditions and $n_{\text{FC}} = n_{\text{EFC}} + n_{\text{OFC}} + n_{\text{PFC}}$ feedback conditions. The total number of conditions shall be denoted by $n_{\text{ICFC}} = n_{\text{IC}} + n_{\text{FC}}$.

For modelling the CBS, the `StimulationInterface`, the `InhibitionInterface`, and the `ActivationCalculation` of a standard behaviour are reused. Figure 4.19 depicts the `CBSActivityCalculation`. This template does not depend on the number of conditions, i.e. it is static and always possesses 9 locations and 16 edges. The (also static) `CBSTargetRatingCalculation`, which is shown in Fig. 4.20, features 4 locations and 5 edges. By contrast, the `CBSInputChangedInterface` (see Figs. 4.21 and 4.27) is a dynamic template that depends on the number of conditions (i.e. number of connected behaviours). It features one central location. For each connected behaviour, one committed location and two edges are added. Therefore, its total number of locations is $1 + n_{\text{IC}}$ and its total number of edges is $2 \cdot n_{\text{IC}}$ for the instance dealing with the input conditions. The numbers for the instance dealing with the feedback conditions are $1 + n_{\text{FC}}$ and $2 \cdot n_{\text{FC}}$, respectively.

`CBSOrderingFulfilled` (see Fig. 4.22) possesses 3 locations and 5 edges. For its $n_{\text{OIC}} + n_{\text{OFC}}$ instances, this yields $3 \cdot (n_{\text{OIC}} + n_{\text{OFC}})$ locations and $5 \cdot (n_{\text{OIC}} + n_{\text{OFC}})$ edges. `CBSPermanentFulfilled` (depicted in Fig. 4.23) has 4 locations and 7 edges. Therefore, its instances in total have $4 \cdot (n_{\text{PIC}} + n_{\text{PFC}})$ locations and $7 \cdot (n_{\text{PIC}} + n_{\text{PFC}})$ edges. For `CBSEnablingFulfilled` (shown in Fig. 4.24), it shall be assumed that $n_{\text{OIC}} > 0$, $n_{\text{PIC}} > 0$, $n_{\text{OFC}} > 0$, and $n_{\text{PFC}} > 0$, i.e. the most complex version of this template is needed. This version features 19 locations and 49 edges. For all instances, this sums up to $19 \cdot (n_{\text{EIC}} + n_{\text{EFC}})$ locations and $49 \cdot (n_{\text{EIC}} + n_{\text{EFC}})$ edges. `CBSCombineEnabling`, `CBSCombineOrdering`, and `CBSCombinePermanent` are templates with the basic structure of a hypercube. Again, the edges are bidirectional. For each additional condition, two locations and two edges have to be added. Furthermore, from each of the non-committed locations, a `reset` edge leads to the initial location. In the case of `CBSCombineEnabling` (see Fig. 4.28) for the input conditions, this yields $2^{n_{\text{EIC}}} + 2 \cdot n_{\text{EIC}}$ locations and $2 \cdot 2^{n_{\text{EIC}}-1} \cdot n_{\text{EIC}} + 2 \cdot n_{\text{EIC}} + 2^{n_{\text{EIC}}} - 1 = 2^{n_{\text{EIC}}} \cdot (n_{\text{EIC}} + 1) + 2 \cdot n_{\text{EIC}} - 1$ edges. For the other conditions, these numbers can be calculated analogously. `CBSConditionsFulfilled` (see Fig. 4.25) is a static template (as-

suming that conditions of all three types exist) with 14 locations and 37 edges per instance. The last template needed for modelling the CBS, **CBSConnectICAndFC** (see Fig. 4.26), also depends on the number of conditions. It has $4 + n_{\text{ICFC}}$ locations and $5 + n_{\text{ICFC}}$ edges. Hence, the number of locations of the CBS-specific templates is as follows:

$$\begin{aligned}
& 9 + 4 + 1 + n_{\text{IC}} + 1 + n_{\text{FC}} \\
& + 3 \cdot (n_{\text{OIC}} + n_{\text{OFC}}) + 4 \cdot (n_{\text{PIC}} + n_{\text{PFC}}) + 19 \cdot (n_{\text{EIC}} + n_{\text{EFC}}) \\
& + 2^{n_{\text{EIC}}} + 2 \cdot n_{\text{EIC}} + 2^{n_{\text{OIC}}} + 2 \cdot n_{\text{OIC}} + 2^{n_{\text{PIC}}} + 2 \cdot n_{\text{PIC}} \\
& + 2^{n_{\text{EFC}}} + 2 \cdot n_{\text{EFC}} + 2^{n_{\text{OFC}}} + 2 \cdot n_{\text{OFC}} + 2^{n_{\text{PFC}}} + 2 \cdot n_{\text{PFC}} + 14 + 4 + n_{\text{ICFC}} \\
= & 2^{n_{\text{EIC}}} + 23 \cdot n_{\text{EIC}} + 2^{n_{\text{OIC}}} + 7 \cdot n_{\text{OIC}} + 2^{n_{\text{PIC}}} + 8 \cdot n_{\text{PIC}} \\
& + 2^{n_{\text{EFC}}} + 23 \cdot n_{\text{EFC}} + 2^{n_{\text{OFC}}} + 7 \cdot n_{\text{OFC}} + 2^{n_{\text{PFC}}} + 8 \cdot n_{\text{PFC}} + 33
\end{aligned}$$

The number of edges of the CBS-specific templates is:

$$\begin{aligned}
& 16 + 5 + 2 \cdot n_{\text{IC}} + 2 \cdot n_{\text{FC}} \\
& + 5 \cdot (n_{\text{OIC}} + n_{\text{OFC}}) + 7 \cdot (n_{\text{PIC}} + n_{\text{PFC}}) + 49 \cdot (n_{\text{EIC}} + n_{\text{EFC}}) \\
& + 2^{n_{\text{EIC}}} \cdot (n_{\text{EIC}} + 1) + 2 \cdot n_{\text{EIC}} - 1 + 2^{n_{\text{OIC}}} \cdot (n_{\text{OIC}} + 1) + 2 \cdot n_{\text{OIC}} - 1 \\
& + 2^{n_{\text{PIC}}} \cdot (n_{\text{PIC}} + 1) + 2 \cdot n_{\text{PIC}} - 1 + 2^{n_{\text{EFC}}} \cdot (n_{\text{EFC}} + 1) + 2 \cdot n_{\text{EFC}} - 1 \\
& + 2^{n_{\text{OFC}}} \cdot (n_{\text{OFC}} + 1) + 2 \cdot n_{\text{OFC}} - 1 + 2^{n_{\text{PFC}}} \cdot (n_{\text{PFC}} + 1) + 2 \cdot n_{\text{PFC}} - 1 \\
& + 37 + 5 + n_{\text{ICFC}} \\
= & 2^{n_{\text{EIC}}} \cdot (n_{\text{EIC}} + 1) + 54 \cdot n_{\text{EIC}} + 2^{n_{\text{OIC}}} \cdot (n_{\text{OIC}} + 1) + 10 \cdot n_{\text{OIC}} \\
& + 2^{n_{\text{PIC}}} \cdot (n_{\text{PIC}} + 1) + 12 \cdot n_{\text{PIC}} + 2^{n_{\text{EFC}}} \cdot (n_{\text{EFC}} + 1) + 54 \cdot n_{\text{EFC}} \\
& + 2^{n_{\text{OFC}}} \cdot (n_{\text{OFC}} + 1) + 10 \cdot n_{\text{OFC}} + 2^{n_{\text{PFC}}} \cdot (n_{\text{PFC}} + 1) + 12 \cdot n_{\text{PFC}} + 57
\end{aligned}$$

Table 4.1 summarises the results of this section about the quantitative aspects of the proposed UPPAAL models. With this table, it is possible to estimate the state space needed for the model of an arbitrary behaviour network. A number of improvements have been made to the algorithm which automatically creates a UPPAAL model of an iB2C network with the aim to reduce the complexity of the resulting models. These improvements are technical details that shall not be discussed here. However, they only reduce the amount of locations and edges that are needed in certain cases. Hence, the numbers in Tab. 4.1 can be regarded as the worst case.

With the results of this section, iB2C networks of arbitrary complexity can be modelled as networks of UPPAAL automata. Figure 4.30 depicts the integration of this aspect into the overall concept. Naturally, the creation of a formal model of an iB2C network is only the first of two steps necessary for the verification of the network. The second step—the actual verification—is presented in the next section.

Table 4.1: The numbers of locations and edges of each template (n_i : number of inhibiting behaviours; n_c : number of competing behaviours; n_{EIC} , n_{OIC} , n_{PIC} , n_{EFC} , n_{OFC} , n_{PFC} : number of enabling, ordering, and permanent input and feedback conditions; $n_{IC} = n_{EIC} + n_{OIC} + n_{PIC}$; $n_{FC} = n_{EFC} + n_{OFC} + n_{PFC}$; $n_{ICFC} = n_{IC} + n_{FC}$).

Template	#Locations	#Edges
StimulationInterface	4	4
InhibitionInterface	$2^{n_i} + 2 \cdot n_i$	$n_i \cdot (2^{n_i} + 2)$
ActivationCalculation	4	7
ActivityCalculation	4	6
TargetRatingCalculation	2	2
FBActivityCalculation (V. 1)	$2^{(n_c+2)} + 2 \cdot n_c - 2$	$2^{n_c+1} \cdot (n_c + 2) + 2 \cdot n_c - 2$
FBIBActivityChanged	8	12
FBActivityCalculation (V. 2)	$2^{n_c} + 2 \cdot n_c$	$n_c \cdot (2^{n_c} + 2)$
FBTargetRatingCalculation	$2^{n_c} + 2 \cdot n_c$	$n_c \cdot (2^{n_c} + 2)$
CBSInputChangedInterface	$1 + n_{IC}$	$2 \cdot n_{IC}$
	$1 + n_{FC}$	$2 \cdot n_{FC}$
CBSEnablingFulfilled	$19 \cdot n_{EIC}$	$49 \cdot n_{EIC}$
	$19 \cdot n_{EFC}$	$49 \cdot n_{EFC}$
CBSOrderingFulfilled	$3 \cdot n_{OIC}$	$5 \cdot n_{OIC}$
	$3 \cdot n_{OFC}$	$5 \cdot n_{OFC}$
CBSPermanentFulfilled	$4 \cdot n_{PIC}$	$7 \cdot n_{PIC}$
	$4 \cdot n_{PFC}$	$7 \cdot n_{PFC}$
CBSCombineEnabling	$2^{n_{EIC}} + 2 \cdot n_{EIC}$	$2^{n_{EIC}} \cdot (n_{EIC} + 1) + 2 \cdot n_{EIC} - 1$
	$2^{n_{EFC}} + 2 \cdot n_{EFC}$	$2^{n_{EFC}} \cdot (n_{EFC} + 1) + 2 \cdot n_{EFC} - 1$
CBSCombineOrdering	$2^{n_{OIC}} + 2 \cdot n_{OIC}$	$2^{n_{OIC}} \cdot (n_{OIC} + 1) + 2 \cdot n_{OIC} - 1$
	$2^{n_{OFC}} + 2 \cdot n_{OFC}$	$2^{n_{OFC}} \cdot (n_{OFC} + 1) + 2 \cdot n_{OFC} - 1$
CBSCombinePermanent	$2^{n_{PIC}} + 2 \cdot n_{PIC}$	$2^{n_{PIC}} \cdot (n_{PIC} + 1) + 2 \cdot n_{PIC} - 1$
	$2^{n_{PFC}} + 2 \cdot n_{PFC}$	$2^{n_{PFC}} \cdot (n_{PFC} + 1) + 2 \cdot n_{PFC} - 1$
CBSConditionsFulfilled	14	37
CBSConnectICAndFC	$4 + n_{ICFC}$	$5 + n_{ICFC}$
CBSActivityCalculation	9	16
CBSTargetRatingCalculation	4	5

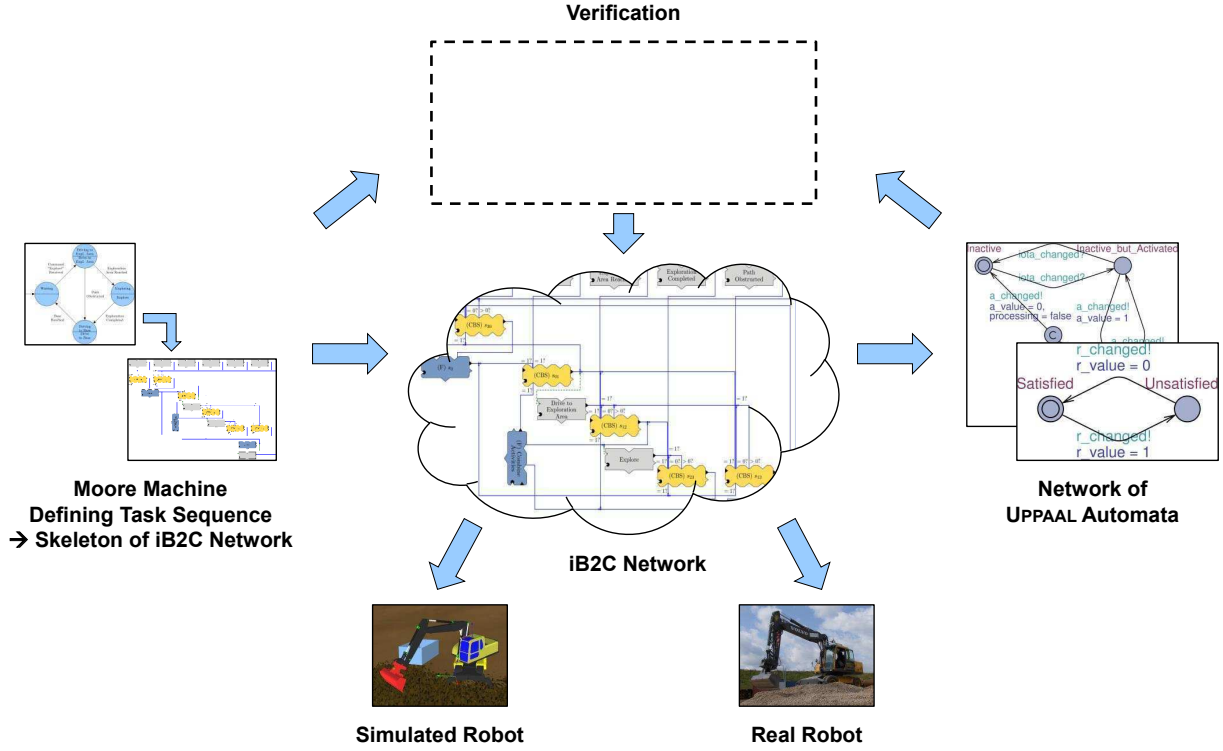


Figure 4.30: The modelling of iB2C networks as networks of UPPAAL automata has been added to the overall concept of this doctoral thesis.

4.3 Verifying iB2C Networks Using Model Checking

After an iB2C network has been modelled as a network of UPPAAL automata in the way described in Sec. 4.2, UPPAAL's verifier can be used to check certain properties of the model. These properties often target the circumstances under which a certain behaviour or group of behaviours can get active, but other aspects can also occur. In order to employ UPPAAL's model checking capabilities, the properties in question have to be translated into queries that UPPAAL's verifier can process. Some information about the query language (a simplified version of TCTL) has already been provided in Sec. 4.1.2.2.

A simple example that is based on Fig. 4.13 (A behaviour B_0 stimulates a behaviour B_1 with its activity.) shall be given in the following. The involved templates are depicted in Figs. 4.7, 4.9, and 4.10. A query shall be used to determine whether B_1 can be active without B_0 being active, too. In other words, it shall be determined whether $B1_activity_calculation$ can be in location *Active* without $B0_activity_calculation$ being in location *Active*. This can be translated into the following query:

```
E<> (B1_activity_calculation.Active && !B0_activity_calculation.Active)
```

According to the iB2C principle mentioned in Sec. 2.2.1, the activity a_B of a behaviour B is limited by the behaviour's activation ι_B : $a_B \leq \iota_B$. Hence, the query should be evaluated to false. However, the verifier evaluates it to true. The reason for this lies in the intermediate locations, which have already been mentioned in Sec. 4.2.1: If $B0_activity_calculation$ enters location *Active*, $B1_activation_calculation$ can consequently enter location *Activated*. This will result in $B1_activity_calculation$

transition to location `Inactive_but_Activated`. After that, it can switch to `Active`. If `B0_activity_calculation` leaves `Active`, it will enter the intermediate location before proceeding to `Inactive`. In this moment, the above query will be evaluated to true. But the state of the model does not correspond to a state of the behaviour network which it originates from. As has been explained in Sec. 4.2.1, the `processing` flag has been added to exclude the intermediate locations from the results of queries. The adapted query is the following:

```
E<> (B1_activity_calculation.Active && !B0_activity_calculation.Active &&
!Processing())
```

This query is evaluated to false. `Processing()` is a function that returns true if and only if the `Processing` flag of at least one of the automata of the system is set. It is created automatically during the creation of the UPPAAL model. In the interest of greater clarity, the check of the `Processing` flag will be left out of the queries in the remainder of this thesis.

A simple query with a universal quantifier can be used to disprove that a high activity of B_0 implies a high activity of B_1 , i.e. to disprove that $(a_{B_0} = 1) \implies (a_{B_1} = 1)$. $(a_{B_0} = 1) \implies (a_{B_1} = 1)$ is equivalent to $(!(a_{B_0} = 1)) \vee (a_{B_1} = 1)$. Applying the path quantifier A and the temporal operator \Box (cp. Sec. 4.1.2) yields the following query:

```
A[] (B1_activity_calculation.Active || !B0_activity_calculation.Active)
```

This query is evaluated to false, which is correct as B_1 can be stimulated by B_0 , but still be inactive, i.e. $(a_{B_0} = 1) \not\Rightarrow (a_{B_1} = 1)$.

The following section will provide examples of more complex queries used to analyse a part of the robot control system of the off-road vehicle RAVON.

4.3.1 Example Application: Navigation System of RAVON

The autonomous off-road vehicle RAVON (see Fig. 2.20) has already been presented in Sec. 2.2.1 as an example of a robot whose control system contains numerous iB2C behaviours. In Sec. 3.1.2, it has been explained how behaviour activity sequences implemented in RAVON's control network can be used to realise turning manoeuvres and detect dead ends, respectively. In this section, another part of the navigation system shall be analysed using the presented verification approach.

In RAVON's control system, the iB2C behaviour group *(G) Drive Control* (see Fig. 4.31) takes care of coordinating several (groups of) behaviours that realise different approaches for calculating target coordinates. These coordinates are used for what is called *point access* in the control system of RAVON—driving towards a goal. The *(G) Drive Control* shall be verified against a number of requirements using the approach presented above (cp. [Armbrust 12a]).

Basically, there are two ways of making RAVON drive towards a goal:

1. a *direct point access*, which guides the robot directly to a target location
2. a *point access with orientation*, which extends the direct point access by additionally specifying the robot's desired orientation at the target

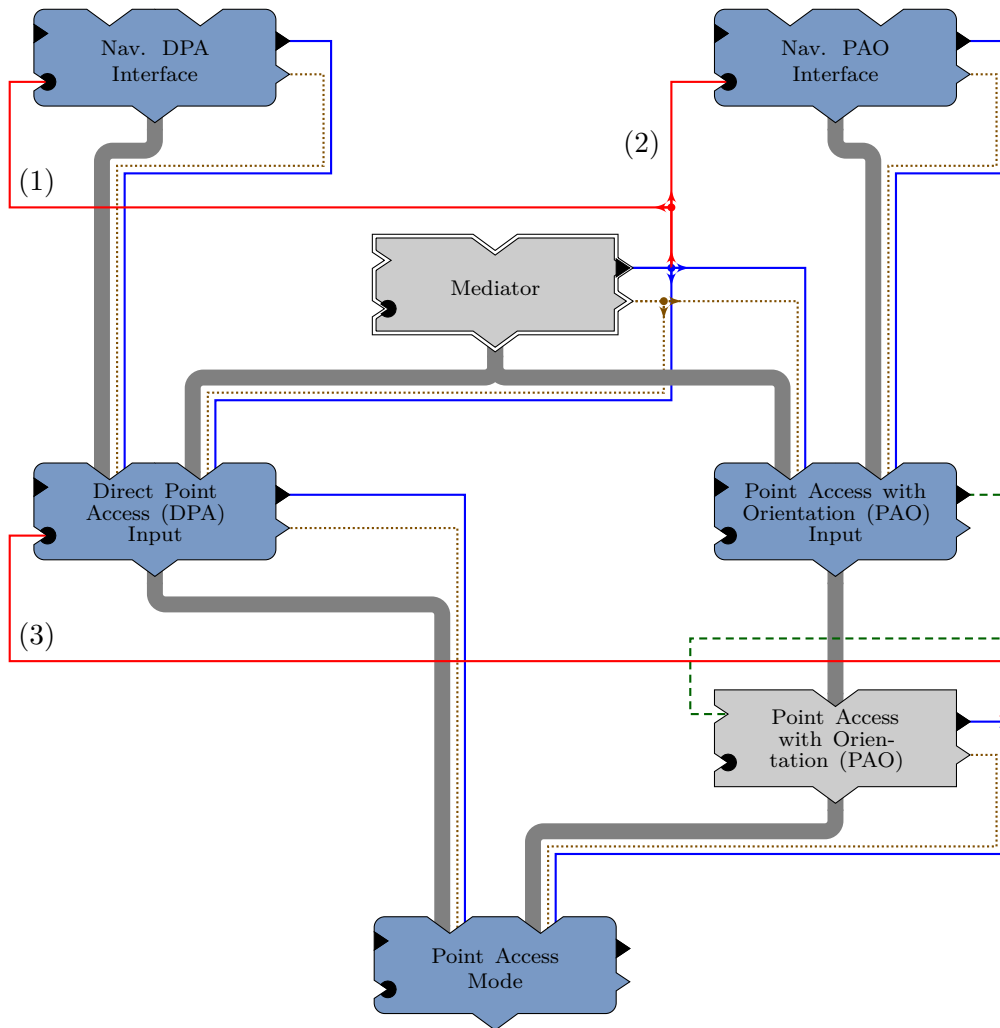


Figure 4.31: The (G) Drive Control, a behaviour group within the navigation system of RAVON that contains different components providing target coordinates for a point access behaviour.

There are currently three navigation approaches calculating target coordinates, which are coordinated in the *(G) Drive Control*:

1. A higher-level navigation component is able to provide target coordinates (with or without a target orientation) via the two behaviours *(F) Navigator Point Access with Orientation Interface* and *(F) Navigator Direct Point Access Interface*, i.e. these two behaviours serve as an interface for the higher-level navigation component.
2. A classic path planner (*(G) Local Path Planner*) uses the A* algorithm to plan paths on local grid maps and is able to provide target coordinates.
3. A set of components that detects special structures in the environment—so-called *passages* (see below)—is able to provide target coordinates along with the desired orientation at the target.

The higher-level navigation component in RAVON's control system has not been designed in a behaviour-based fashion. Hence, an interface to the behaviour-based components is necessary. This is built by *(F) Navigator Direct Point Access Interface ((F) Nav. DPA Interface)* and *(F) Navigator Point Access with Orientation Interface ((F) Nav. PAO Interface)*.

The classic path planner and the components dealing with passages have been combined in the behaviour group *(G) Mediator* (see Fig. 4.32). As the path planner comprises several components, it has also been realised as a behaviour-based group. Passages have already been mentioned in Secs. 3.1.2.1 and 3.1.2.2. They are paths leading through obstacle formations in the robot's environment and are detected in the control system of RAVON using abstract views on the environment referred to as *virtual sensors* and *virtual sensor probes*. How this is done is described in detail in [Armbrust 09b] and [Schäfer 11]. The components realising the passage-based approach are marked with a grey background in Fig. 4.32. They work as follows: *New Passage (NP)* gets active if a new passage is detected and sends the coordinates of the passage to the *Passage Manager (PM)*. The latter reasons about whether RAVON should enter the passage. If this is the case, the *Passage Manager* gets active and forwards the coordinates of the passage to *(F) Passage Driver Target ((F) PDT)*. *Same Passage (SP)* checks whether the currently detected passage is the same as the previously detected one. The behaviour's intention is to guide the robot towards the current target passage as long as the passage is detected by the robot's sensor system. If this is the case, *Same Passage* gets active and forwards the coordinates of the passage to *(F) Passage Driver Target*. The latter performs a maximum fusion of the target coordinates provided by the *Passage Manager* and *Same Passage* and sends the result to the *Passage Driver (PD)*, which further processes the coordinates. The behaviour's output is transmitted to *(F) Mediator*, a fusion behaviour that performs a maximum fusion of the outputs of the *Passage Driver* and the *(G) Local Path Planner ((G) LPP)*. *(F) Mediator*, being the coordinating fusion behaviour of *(G) Mediator*, sends the result to the output ports of the containing group.

The fusion behaviours *(F) Direct Point Access Input ((F) DPA Input)* and *(F) Point Access with Orientation Input ((F) PAO Input)* combine target coordinates without and with target orientation, respectively (see Fig. 4.31). The former forwards the target coordinates

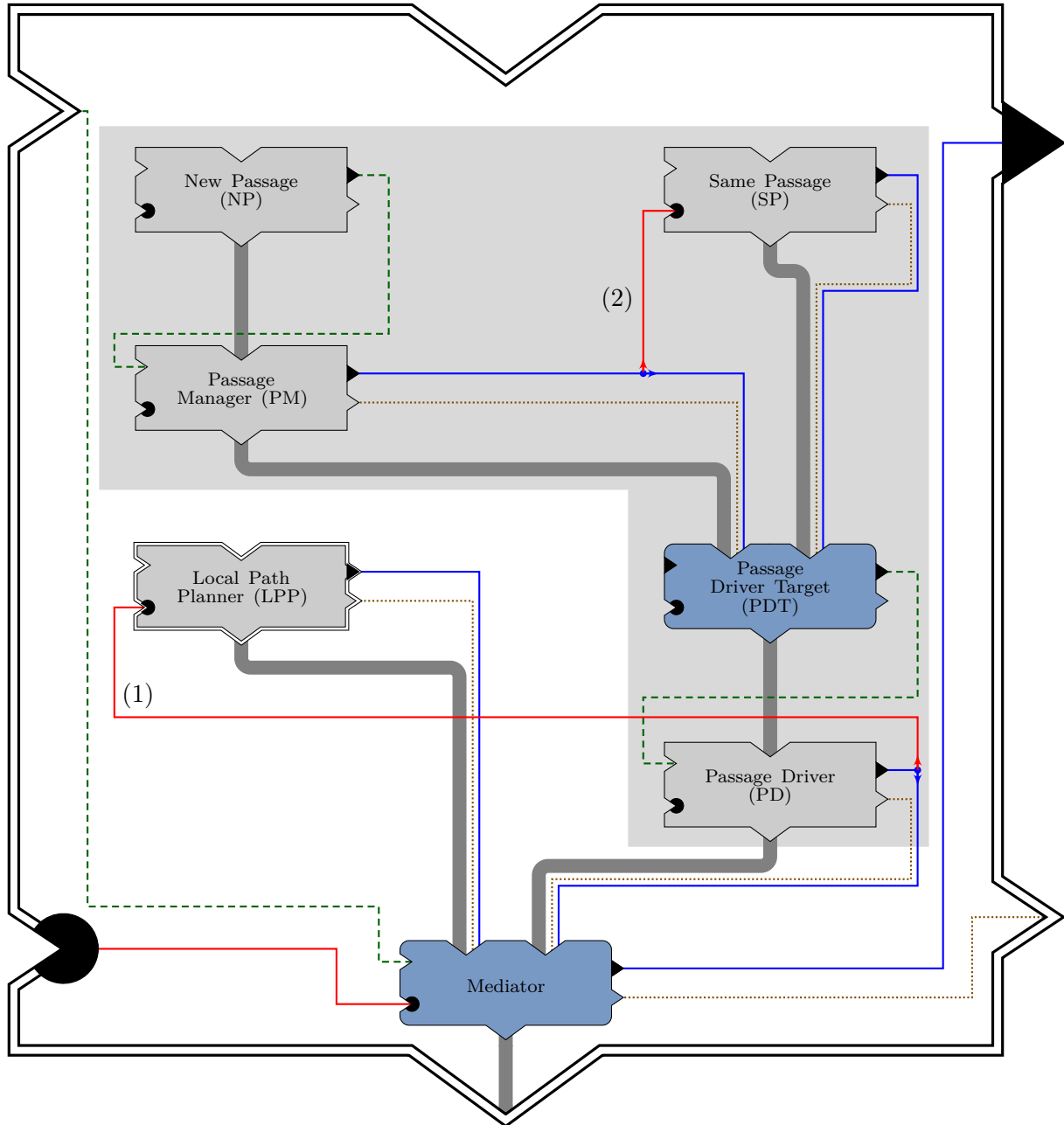


Figure 4.32: The *(G) Mediator*, a behaviour group within the navigation system of RAVON that combines a passage-based approach with a classic A*-based path planner. The components realising the passage-based approach are marked with a grey background.

directly to another fusion behaviour (*(F) Point Access Mode*), while the latter sends them—along with the desired orientation at the target—to *Point Access with Orientation (PAO)*, where intermediate target points are calculated that guide the robot in such a way to the final target that it reaches the target with the desired orientation. These intermediate target points are then sent down to *(F) Point Access Mode*.

The requirements to be verified originate from the fact that at any point in time, only one of the three groups of behaviours (*(F) Nav. DPA Interface* and *(F) Nav. PAO Interface*; *(G) Local Path Planner*; *New Passage*, *Same Passage*, and *Passage Manager*) realising one of the above-mentioned navigation approaches shall be able to send target coordinates to the lower layer. Moreover, the usage of a newly detected passage shall be preferred to the usage of the current passage. Furthermore, if the execution of a point access with orientation is possible, it shall be preferred to the execution of a direct point access. For this purpose, the following three orders of precedence have been defined:

1. $NP, PM, SP \prec (G) LPP \prec (F) Nav. DPA Interface, (F) Nav. PAO Interface$
2. $NP, PM \prec SP$
3. $PAO \prec (F) DPA Input$

$B_0 \prec B_1$ means that B_0 has precedence over B_1 , which is the case if the following two statements are fulfilled:

1. If B_0 is active, then B_1 is not activated, i.e. $(a_{B_0} = 1) \implies (\iota_{B_1} = 0)$.
2. B_0 can be activated even if B_1 is active, i.e. $(a_{B_1} = 1) \not\Rightarrow (\iota_{B_0} = 0)$.

The idea is that B_0 inhibits B_1 with its activity and that B_1 cannot completely inhibit B_0 . Five requirements can be derived from the above comments about RAVON's navigation system:

- R1) Only one of the three groups of behaviours shall be able to provide target coordinates to the lower layer at the same time.
- R2) The *(G) LPP* shall have precedence over the *(F) Nav. DPA Interface* and the *(F) Nav. PAO Interface*.
- R3) *SP* shall have precedence over the *(G) LPP*.
- R4) *NP* and the *PM* shall have precedence over *SP*.
- R5) The *PAO* shall have precedence over the *(F) DPA Input*.

In the following, it is verified using the previously described concept that the navigation system fulfils these requirements.

Evaluation of Requirement R1

Only the outputs of active behaviours can have an influence on the outcome of a fusion in a fusion behaviour. Hence, only an active behaviour can provide target coordinates to a lower layer. Therefore, it is first of all verified that the *(G) Mediator* cannot be active at the same time as one of the two behaviours *(F) Nav. DPA Interface* and *(F) Nav. PAO Interface*. This is done with the following query:

```
E<> ( (F_Mediator_a_value == 1) && ((F_Nav_DPA_Interface_a_value == 1)
|| (F_Nav_PAO_Interface_a_value == 1)) ) → false
```

This query is correctly evaluated to false, indicating that either the interfaces to the higher-level navigation or the *(G) Mediator* can be active. The reason for this are the inhibitory connections from the activity port of the *(G) Mediator* to the inhibition ports of the *(F) Nav. DPA Interface* and the *(F) Nav. PAO Interface* (see Fig. 4.31 (1) and (2)).

The second task is to verify that at most one of the components within the *(G) Mediator* can decide which target coordinates to send to the lower-level navigation components of RAVON's control system. The following query is evaluated to true, which yields that it is possible that the path planning and the passage components are active concurrently.

```
E<> ( ((SP_a_value == 1) || (NP_a_value == 1)) && (G_LPP_a_value == 1) )
→ true
```

At first sight, this could be interpreted as two of the components being able to send target coordinates downwards, thus violating requirement 1. As the outputs of the two components are combined by *(F) Mediator*, a violation of the requirement can only occur if *(F) Mediator* reaches a state where both input behaviours are active, which corresponds to the location `Active_11` of `F_Mediator_activity_calculation`. With another query, it can be proven that this is never the case:

```
E<> (F_Mediator_activity_calculation.Active_11) → false
```

The reason for this is the connection of the activity output of the *Passage Driver* with the inhibition input of the *(G) Local Path Planner* (see Fig. 4.32 (1)), with which the *Passage Driver* can—by getting active—set the activation of the *(G) Local Path Planner* to 0, which means that the *(G) Local Path Planner* cannot get active.

In summary, requirement 1) has been verified, meaning that only one of the three groups of behaviours is able to provide target coordinates to the lower layer at the same time.

If the above course of action is continued, another aspect can be verified, namely that either *Same Passage* or *New Passage* can provide target coordinates, but not both at the same time. The result of the query

```
E<> ( (SP_a_value == 1) && (NP_a_value == 1) ) → true
```

seems to indicate that this is not the case, i.e. that it is possible that the outputs of both passage behaviours are propagated further down to *(F) Passage Driver Target* and from there to the *Passage Driver*. However, the target coordinates provided by *New Passage* are not directly sent to *(F) Passage Driver Target*, but first to the *Passage Manager*, which can forward them to the *Passage Driver*. Hence, it has to be checked whether *Same Passage* and the *Passage Manager* can get active at the same time. This can be done with the following query:

```
E<> ( (SP_a_value == 1) && (PM_a_value == 1) ) → false
```

As the result of the query shows, either *Same Passage* or the *Passage Manager* can be active, but not both at the same time. Similar to the *Passage Driver* and the *(G) Local Path Planner*, an inhibitory link from the *Passage Manager* to *Same Passage* takes care of this (see Fig. 4.32 (2)). Therefore, either the outputs of *Same Passage* or the ones of *New Passage* are sent to the lower layers, although both behaviours can be active at the same time.

The above results might seem very abstract, but they have a concrete meaning in practice. RAVON will only operate in a sensible way if not more than one of the behaviour groups can exercise control over its actuators. There is no sense in sending, for example, the target calculated by the local path planner and the one calculated by the passage components at the same time as input to the point access behaviours. As has been shown above, the proposed verification concept can be used to prove that the developed iB2C network has been built correctly with respect to this aspect.

Evaluation of Requirement R2

In order to show that the *(G) Local Path Planner* has precedence over the *(F) Nav. DPA Interface* and the *(F) Nav. PAO Interface*, it first has to be proven that if the *(G) Local Path Planner* is active, then the *(F) Nav. DPA Interface* and the *(F) Nav. PAO Interface* are not activated, i.e.:

$$(a_{(G) \text{ LPP}} = 1) \implies (\iota_{(F) \text{ Nav. DPA Interface}} = 0 \wedge \iota_{(F) \text{ Nav. PAO Interface}} = 0)$$

This can be achieved using the following query, which is evaluated to false:

```
E<> ( (F_Nav_DPA_Interface_activation_calculation.Activated
|| F_Nav_PAO_Interface_activation_calculation.Activated)
&& (G_LPP_a_value == 1) ) → false
```

Second, it has to be shown that the *(G) Local Path Planner* can be activated even if the *(F) Nav. DPA Interface* or the *(F) Nav. PAO Interface* is active, i.e.:

$$(a_{(F) \text{ Nav. DPA Interface}} = 1 \vee a_{(F) \text{ Nav. PAO Interface}} = 1) \not\Rightarrow (\iota_{(G) \text{ LPP}} = 0)$$

This can be done with the help of two queries:

```
E<> ( (F_Nav_DPA_Interface_a_value == 1)
&& (G_LPP_activation_calculation.Activated) ) → true
E<> ( (F_Nav_PAO_Interface_a_value == 1)
&& (G_LPP_activation_calculation.Activated) ) → true
```

Both of them are evaluated to true. The evaluation of the three above queries proves that in fact, the *(G) Local Path Planner* has precedence over the *(F) Nav. DPA Interface* and the *(F) Nav. PAO Interface*. The reason for this are the inhibitory links from the *(G) Mediator* to the *(F) Nav. DPA Interface* and the *(F) Nav. PAO Interface* (see Fig. 4.31 (1) and (2)). As the *(G) Local Path Planner* can influence the activity of the *(G) Mediator*, it can use these links to inhibit the *(F) Nav. DPA Interface* and the *(F) Nav. PAO Interface*.

Hence, requirement 2) has also been verified. In practice, this means that the *(G) Local Path Planner* can overwrite the target provided by the higher navigation layer with

intermediate target coordinates calculated using the A* algorithm. The final target of the robot will then still be the one provided by the high-level navigation, but the ability of the *(G) Local Path Planner* to plan paths around obstacles in the robot's vicinity will be taken advantage of. Using model checking, it could be proven that RAVON's control system supports this approach.

Evaluation of Requirement R3

First, it has to be checked whether the *(G) Local Path Planner* is never activated if *Same Passage* is active. However, the following query is evaluated to true, indicating that the *(G) Local Path Planner* can be activated if *Same Passage* is active:

```
E<> ( (G_LPP_activation_calculation.Activated) && (SP_a_value == 1) )
→ true
```

The reason is that there is no direct inhibitory link from *Same Passage* to the *(G) Local Path Planner*. Instead, an inhibitory link connects the activity port of the *Passage Driver* with the inhibition port of the *(G) Local Path Planner* (see Fig. 4.32 (1)). This is sufficient as the target coordinates of *Same Passage* have to pass the *Passage Driver* on their way to the lower layers. Hence, *Same Passage* is able to stimulate the *Passage Driver* via *(F) Passage Driver Target*. Therefore, it is sufficient to check whether the *Passage Driver* has precedence over the *(G) Local Path Planner*. This is achieved with the following two queries:

```
E<> ( (G_LPP_activation_calculation.Activated) && (PD_a_value == 1) )
→ false
```

```
E<> ( (G_LPP_a_value == 1) && (PD_activation_calculation.Activated) )
→ true
```

The practical effect of requirement 3) is that when RAVON's sensor processing is currently detecting a suitable passage, the robot will ignore the output of the *(G) Local Path Planner* and drive towards the passage. If the verification had yielded that the corresponding precedence ($SP \prec (G) LPP$) was not correctly implemented in the control system, this would mean that it could happen that the *(G) Local Path Planner* would guide the robot despite *Same Passage* being able to provide a better target.

Evaluation of Requirement R4

Similar to requirement 3, the first check is whether *Same Passage* is never activated if *New Passage* or the *Passage Manager* is active. The two queries are the following:

```
E<> ( (SP_activation_calculation.Activated) && (NP_a_value == 1) ) → true
```

```
E<> ( (SP_activation_calculation.Activated) && (PM_a_value == 1) ) → false
```

In order for requirement 4 to be fulfilled, both queries should be evaluated to false. However, the first query is evaluated to true as there is no direct inhibitory link from *New Passage* to *Same Passage*. Instead, there is such a link from the *Passage Manager* to *Same Passage* (see Fig. 4.32 (2)), resulting in the second query to be evaluated to false. The reason is—similar to requirement 3—that *New Passage* sends its target coordinates via the *Passage Manager* downwards. Therefore, it is able to stimulate the *Passage Manager*. Hence, the inhibitory connection between the *Passage Manager* and *Same Passage* is

sufficient. Naturally, it has to be checked whether the *Passage Manager* can be activated even in case *Same Passage* is active. The corresponding query is—correctly—evaluated to true:

```
E<> ( (SP_a_value == 1) && (PM_activation_calculation.Activated) ) → true
```

The result of this verification step is that RAVON's navigation system prefers a newly detected passage over the passage that the robot is currently driving to.

Evaluation of Requirement R5

The purpose of this requirement is to ensure that a point access with target orientation is always executed if possible. Therefore, the *Point Access with Orientation* needs to have precedence over the *(F) Direct Point Access Input*. With two queries, this can be verified:

```
E<> ( (F_DPA_Input_activation_calculation.Activated)
&& (PA0_a_value == 1) ) → false
```

```
E<> ( (F_DPA_Input_a_value == 1)
&& (PA0_activation_calculation.Activated) ) → true
```

The results are correct, i.e. the *Point Access with Orientation* indeed has precedence over *(F) Direct Point Access Input*. The reason is the inhibitory connection from *Point Access with Orientation* to *(F) Direct Point Access Input* (see Fig. 4.31 (3)). In other words, in case a point access with orientation can be performed, the navigation system will prefer this to the direct point access.

4.3.2 Tool-Assisted Verification of Behaviour Networks

Section 4.3.1 has demonstrated how the proposed verification technique can be applied. It is possible to make the software automatically create a UPPAAL model of an iB2C network, which can be verified using queries. However, the approach so far does not support the process of creating such queries. While this does not represent a major challenge in the case of simple queries like the ones given in Sec. 4.3.1, the creation of more complex ones is tedious at best, but can easily get error-prone. The aim to assist the developer during the verification process leads to another design decision:

Design Decision 9

To assist the developer during the verification process, graphical tool support for entering queries shall be available.

Another challenge is that UPPAAL is limited with regard to the structure of the supported types of queries. In Sec. 4.1.2.2, the range of queries supported by UPPAAL has been presented. In order to circumvent these limitations, observer automata can be used. The following definition is based on [Behrmann 06]:

Definition 4.10: Observer Automaton

An *observer automaton* is an add-on automaton in charge of detecting events without changing the observed system.

In order to detect a specific state or a sequence of transitions in a system of finite-state automata without having to generate complex queries, an observer automaton can be added to the system. This observer automaton is designed in a way that it transitions to a specific location if the remainder of the system has reached the state in question or if the desired sequence of transitions has occurred. As a result, the query to be generated can usually be much simpler, as it only has to check whether the observer automaton has reached the specific location. The use of observer automata is a common approach and described in many places in the literature, e.g. [Blom 05]. Naturally, the use of an observer automaton moves a part of the complexity of the query generation to the generation of a corresponding observer automaton—which is already an advantage due to UPPAAL’s GUI allowing for the convenient creation of automata. Based upon these considerations, another design decision can be formulated:

Design Decision 10

To assist the developer during the verification process, an automatic transformation of complex queries into a combination of simpler queries and observer automata shall be available.

When using model checking, often not only the result of the model checking process is interesting, but also which trace leads to a counterexample (in case the query is evaluated to false) or to a witness (in case the query is evaluated to true). Reading long traces can be a tedious task. UPPAAL allows for loading a trace and replaying it with its simulator. However, this only shows the transitions of the automata, i.e. the changes in the model. For the developer of a BBS, it is much more helpful to inspect the actual changes of the behaviour network from which the model was created in an appropriate visualisation. This aspect forms the demand of a further design decision:

Design Decision 11

To facilitate the interpretation of the result of a verification process, tool support shall allow for displaying traces created during the verification process as changes in a visualisation of the behaviour network in question.

An approach that allows for graphically creating complex queries that are automatically transformed to simpler queries and observer automata is presented in [Ropertz 12] and [Armbrust 13b]. Its essential aspects will be described in Secs. 4.3.2.1 to 4.3.2.3. In Sec. 4.3.2.4, a FINSTRUCT widget will be presented that meets the demand of Design Decision 11.

4.3.2.1 Properties of Behaviour-Based Systems

Before developing tool support for the generation of complex queries, it has to be decided which properties of BBS shall be proven with the help of model checking and which kind of notation shall be used to define these properties. Defining properties using a very formal notation can make them hard to understand for developers who are inexperienced in formal verification. This led to the aim of finding a definition which is comprehensible to developers with little to no experience in verification.

Approaches that target the solving of this problem by using patterns are presented in [Dwyer 98], [Dwyer 99], and [Meolic 01], for example. These patterns describe frequently used properties informally and provide definitions in formal notations, e.g. in CTL. In contrast to informal descriptions, these formulae can be used as input to model checkers. With the same aim, the author of [Holt 99] has developed a system that translates English specification sentences to formulae of temporal logic. A major disadvantage of approaches which are based on natural languages is that ambiguities which are typical of natural languages complicate the translation to temporal logic formulae.

Another approach is suggested in [Beyer 04]: the use of a plug-in that integrates the model checker BLAST into the Eclipse development environment. The authors claim that with this plug-in, a developer without any knowledge of model checking or formal notations can perform some typical program analysis techniques. While the plug-in facilitates the application of model checking by providing a graphical interface, the user is limited to the types of analyses provided by the plug-in.

The work at hand shares some similarities with the above-mentioned approaches: The verification techniques shall be integrated into FINSTRUCT in order to feature a graphical user interface (cp. Design Decision 9) and a number of properties relevant to the verification of BBS have been developed based on the property patterns described in [Dwyer 99]. In the following, these properties will be presented (see [Armbrust 13b]). Section 4.3.2.2 will then introduce a technique for entering queries graphically based on so-called query graphs.

First, the concept of a “property term” has to be defined:

Definition 4.11: Property Term

Let $bs_B \in \{s_B, i_B, \iota_B, a_B, r_B\}$ be a behaviour signal of behaviour B , $\otimes \in \{<, \leq, =, \geq, >, \neq\}$ a relation symbol, and $st \in \{0, 1\}$ a signal threshold. Then a (basic) *property term* pt is defined as $pt = (bs_B \otimes st)$. Furthermore, if pt_0 and pt_1 are property terms, then $pt_0 \wedge pt_1$ and $pt_0 \vee pt_1$ are also property terms.

Based on these terms, a number of *temporal properties* can be defined:

- `synchronous_before`(pt_{src}, pt_{dst})
- `asynchronous_before`(pt_{src}, pt_{dst})
- `synchronous_paired_before`(pt_{src}, pt_{dst})
- `asynchronous_paired_before`(pt_{src}, pt_{dst})
- `requires_non-strict`(pt_{src}, pt_{dst})
- `requires_strict`(pt_{src}, pt_{dst})
- `synchronous_requires_once`(pt_{src}, pt_{dst})
- `asynchronous_requires_once`(pt_{src}, pt_{dst})
- `globally`(pt)

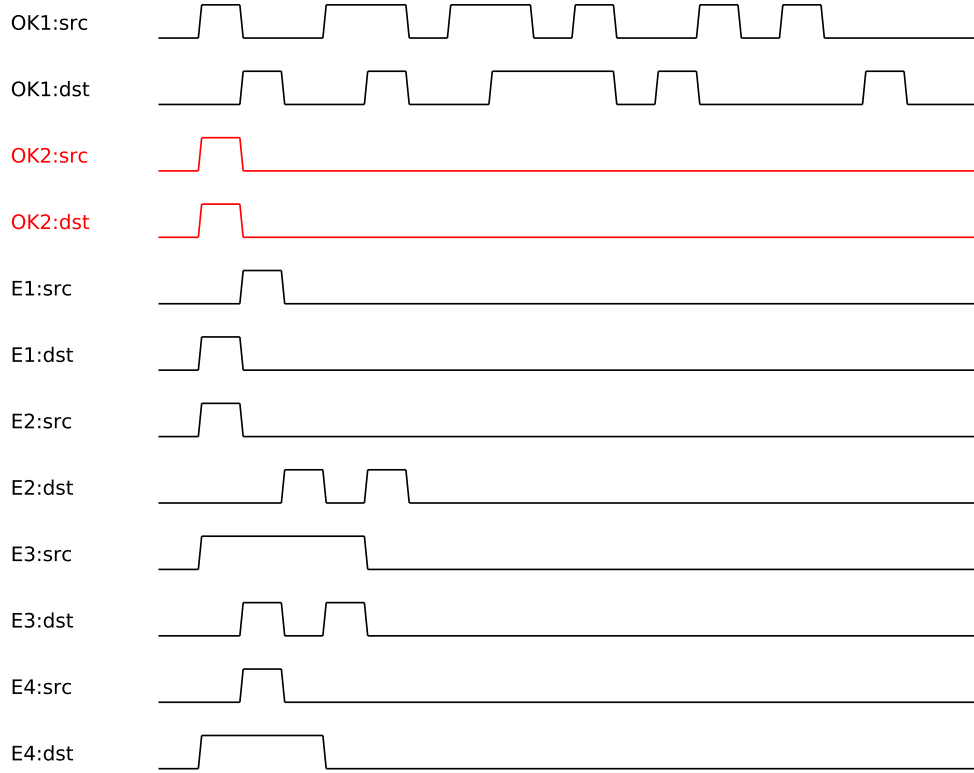


Figure 4.33: A timing diagram illustrating the `synchronous_before` property. The sequences marked in red (OK2:src and OK2:dst) exemplify the difference between `synchronous_before` and `asynchronous_before` (source: [Armbrust 13b]).

- `eventually(pt)`

pt , pt_{src} , and pt_{dst} are property terms, where pt_{src} and pt_{dst} are called *source property term* and *destination property term*, respectively.

The `synchronous_before` property states that pt_{src} has to become true at least once before each occurrence of pt_{dst} . pt_{src} and pt_{dst} are allowed to start being true at the same time, hence the name of the property contains the term “**synchronous**”. By contrast, pt_{dst} and pt_{src} are not allowed to start being true at the same time in case of the property `asynchronous_before`. But like `synchronous_before`, `asynchronous_before` states that pt_{src} has to become true at least once before each occurrence of pt_{dst} .

Figure 4.33 depicts a timing diagram that illustrates the `synchronous_before` property. The diagram shows a number of sequences of the fulfilment of pt_{src} and pt_{dst} . Signals with the suffix “src” in their name refer to pt_{src} , while the ones with the suffix “dst” refer to pt_{dst} . A high value of a signal is interpreted as the corresponding property term being true, while a low value is interpreted as the term being false. If the name of a sequence starts with “OK”, the sequence fulfils the property `synchronous_before`. Correspondingly, the prefix “E” (for “error”) indicates that a sequence does not fulfil the property `synchronous_before`. The two sequences marked in red visualise the difference between `synchronous_before` and `asynchronous_before`: While it is allowed for pt_{src} and pt_{dst} to start being true at the same time (i.e. synchronously) in the case of `synchronous_before`, it is not allowed in the

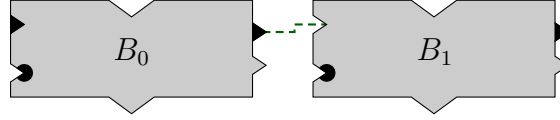


Figure 4.34: A simple iB2C behaviour network in which a behaviour B_0 stimulates a behaviour B_1 .

case of `asynchronous_before`. Hence, the names of the sequences marked in red would have the prefix “E” in the case of `asynchronous_before`, indicating that the property would not be fulfilled for these sequences.

Similar to the `synchronous_before` property, the `synchronous_paired_before` property states that pt_{src} has to start being true before pt_{dst} does. It is allowed that both terms get true at the same time. The difference to `synchronous_before` is that pt_{src} may not start being true more than once between two consecutive occurrences of pt_{dst} being true. In other words, the occurrences of pt_{src} and pt_{dst} becoming true have to appear in pairs. This condition is violated, for example, by the sequences OK1:src and OK1:dst of Fig. 4.33 as pt_{dst} does not get fulfilled during the last two occurrences of pt_{src} getting fulfilled. Like `synchronous_before`, `synchronous_paired_before` has an asynchronous variant, `asynchronous_paired_before`, which differs from `synchronous_paired_before` by the fact that pt_{src} and pt_{dst} are not allowed to start being fulfilled at the same time.

Another property, the `requires` property, also features two variants: `requires_strict` and `requires_non-strict`. Both properties demand that pt_{src} be true in the moment in which pt_{dst} gets true. But while `requires_strict` states that pt_{dst} can only be true as long as pt_{src} is also true, `requires_non-strict` does not demand this.

The properties `synchronous_requires_once` and `asynchronous_requires_once` state that pt_{src} has to be fulfilled at least once before the very first occurrence of pt_{dst} getting fulfilled. Again, the synchronous variant allows for both property terms to start being fulfilled at the same time, while the asynchronous variant does not.

All of the temporal properties that have been mentioned so far are binary and for a meaningful use, pt_{src} has to differ from pt_{dst} . But there are also two unary temporal properties: `globally` and `eventually`. The former demands that the associated property term is always fulfilled, while the latter demands that it is fulfilled at some point in time.

It is also possible to define properties that do not contain a temporal aspect. For example, the property `priority(B_0, B_1)` has been defined for two behaviours B_0 and B_1 as follows: B_0 has a higher priority than B_1 if and only if B_0 can inhibit B_1 and it is possible that B_0 gets active even if B_1 is already active. This property has been introduced previously as precedence in Sec. 4.3.1. It is based on the fact that a behaviour’s activity indicates its influence within a network.

In the following, three small behaviour networks (see Figs. 4.34 to 4.36) shall be used to illustrate some of the queries. In Fig. 4.34, a network of two behaviours B_0 and B_1 is depicted, in which B_0 stimulates B_1 with its activity. The network fulfils the properties `requires_non-strict($a_{B_0} = 1, a_{B_1} = 1$)` and `requires_strict($a_{B_0} = 1, a_{B_1} = 1$)` as B_1 can only get active if it is stimulated by B_0 . This is because of the iB2C principle stating that $a_{B_1} \leq \iota_{B_1}$ (see Sec. 2.2.1). The network depicted in Fig. 4.35 consists of two standard

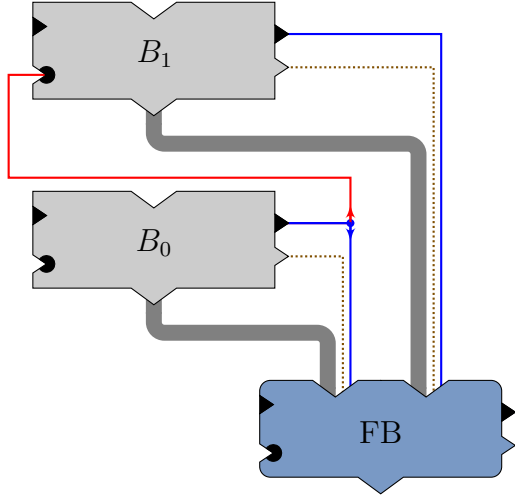


Figure 4.35: A simple iB2C behaviour network in which two behaviours B_0 and B_1 provide inputs to a fusion behaviour. B_0 has a higher priority than B_1 .

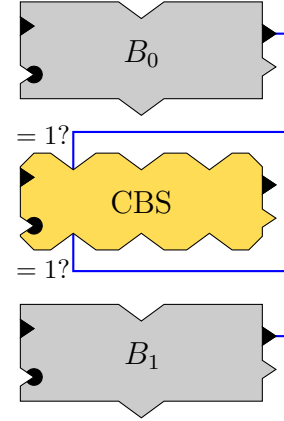


Figure 4.36: A simple iB2C behaviour network in which two behaviours B_0 and B_1 are connected to the enabling input and enabling feedback ports of a CBS, respectively.

behaviours B_0 and B_1 as well as a fusion behaviour FB that combines their outputs. It can be used to illustrate the property $\text{priority}(B_0, B_1)$: Due to the inhibiting connection from B_0 to B_1 , B_0 is able to inhibit B_1 . Furthermore, B_0 can be activated (and then get active) even if B_1 is active. The network also shows the property $\text{eventually}(\iota_{B_1} = 1)$ as B_1 can be activated. However, if B_0 gets active, B_1 gets inactivated. Thus, the network does not feature the property $\text{globally}(\iota_{B_1} = 1)$. An example of a property of the network that refers to the fusion behaviour is $\text{requires_strict}((a_{B_0} = 1) \vee (a_{B_1} = 1), a_{FB} = 1)$ — FB can only get active if one of its input behaviours is active. The third network (see Fig. 4.36) features two standard behaviours B_0 and B_1 as well as a CBS. The activity ports of the standard behaviours are connected to ports of the CBS that are associated with enabling input and feedback conditions, respectively. All three behaviours are permanently stimulated. As a result, B_0 can get active, which will result in the input condition of the CBS being fulfilled. The CBS will then get active and check its feedback condition until B_1 also gets active. At this moment, it will get inactive and check its input condition again. The CBS can only get active if its enabling input condition is fulfilled, i.e. if $a_{B_0} = 1$. Hence, the network has the property $\text{requires_non-strict}(a_{B_0} = 1, a_{CBS} = 1)$. As B_0 does not have to stay active, $\text{requires_strict}(a_{B_0} = 1, a_{CBS} = 1)$ is not fulfilled. If the input condition was permanent, however, $\text{requires_strict}(a_{B_0} = 1, a_{CBS} = 1)$ would also be fulfilled.

The properties described are more complex than the ones used in the example of Sec. 4.3.1 and can prove valuable when verifying a large system. However, the graphical tool support requested by Design Decision 9 and the automatic transformation of complex queries into a combination of simpler queries along with observer automata as requested by Design Decision 10 are still missing. The next section will deal with the request of Design Decision 9 and present an approach to graphically creating complex queries.

4.3.2.2 Graphical Query Design Using Query Graphs

In order to allow for graphically creating queries, a graphical representation of said queries has to be found. Such an approach has been proposed in [Ropertz 12] and [Armbrust 13b] and shall be presented in the following. The approach uses so-called query graphs that graphically represent properties. The following definitions are based on the ones given in [Ropertz 12]:

Definition 4.12: Query Graph

A *query graph* is a direct, asymmetric graph $D = (V, E)$, where V is a set of query vertices and E a set of query edges.

Each query vertex represents a basic property term (see Def. 4.11), a conjunction, or a disjunction and is defined as follows:

Definition 4.13: Query Vertex

A *query vertex* is a triple $v = (B, \text{pt}, \text{type})$, where

- B is a behaviour providing a behaviour signal,
- pt is the basic property term to be represented by v ,
- and type is a flag indicating whether v represents a property term or models a conjunction or disjunction.

Each query edge represents a property (see Sec. 4.3.2.1) or an auxiliary construct and is defined as follows:

Definition 4.14: Query Edge

A *query edge* is a 4-tuple $e = (v_{\text{src}}, v_{\text{dst}}, \text{property}, \text{type})$, where

- v_{src} is the source query vertex,
- v_{dst} is the destination query vertex,
- property is the property to be represented by e ,
- and type is a flag indicating whether e represents a property or is only an auxiliary construct to model a conjunction or disjunction.

The query graph representing the unary temporal property **eventually**($a_{B_1} = 1$) is shown in Fig. 4.37. As a unary temporal property only has one associated property term, the source and destination vertices of the edge representing the property are equal. In Fig. 4.38, the query graph representing the binary temporal property **requires_strict** ($((a_{B_0} = 1) \vee (a_{B_1} = 1), a_{FB} = 1)$) is depicted. This property is one of the properties of the small iB2C network shown in Fig. 4.35. The query graph consists of four query vertices as well as three query edges. The property **requires_strict** is represented by the yellow edge, which connects its source vertex OR with its destination vertex FB. The latter represents the destination property term pt_{dst} , which is $a_{FB} = 1$. The vertex OR does not

directly represent a basic property term. Instead, it models a disjunction of two basic property terms that are represented by two further query vertices, which are connected via auxiliary query edges (visualised by dotted lines). Hence, the source property term pt_{src} is represented by the combination of the query vertices B0 and B1 as well as the connecting vertex OR. Therefore, pt_{src} is $(a_{B_0} = 1) \vee (a_{B_1} = 1)$.

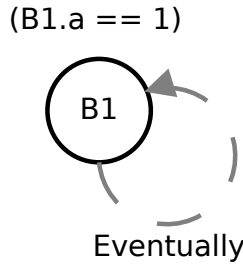


Figure 4.37: The query graph representing the unary temporal property $eventually(a_{B_1} = 1)$ (source: [Armbrust 13b]).

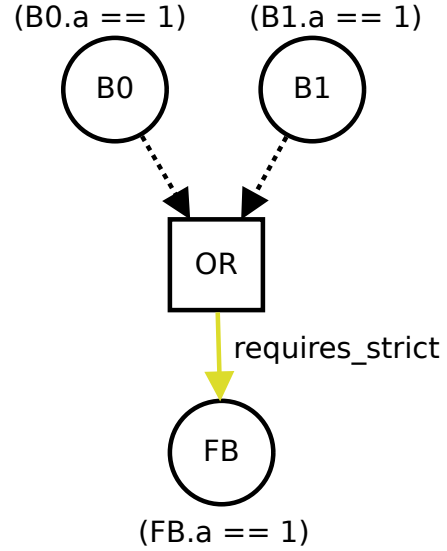


Figure 4.38: The query graph representing the binary temporal property $requires_strict((a_{B_0} = 1) \vee (a_{B_1} = 1), a_{FB} = 1)$ (source: [Armbrust 13b]).

Naturally, a graphical representation of queries alone does not meet the demand for graphical tool support concerning the creation of queries. Hence, a widget has been added to FINSTRUCT that allows for graphically designing query graphs. Figure 4.39 shows this widget. It visualises the two query graphs depicted in Figs. 4.37 and 4.38. This widget meets the demand of Design Decision 9 for the ability to graphically design queries. What is still missing is an automatic transfer from query graphs to real queries (cp. Design Decision 10). The next section will deal with this task.

4.3.2.3 Generation of Queries from Query Graphs

The range of queries UPPAAL supports is limited to the types described in Sec. 4.1.2.2. Hence, not all query graphs can be directly transferred into a query that can be sent to UPPAAL's verifier. As has been mentioned above, observer automata together with reachability and safety properties shall be used to circumvent these limitations. These automata monitor a network of other automata and can transition into dedicated *failure* or *acceptance* locations depending on the state of the remainder of the system. A model of a behaviour network satisfies a property if and only if the corresponding observer automaton never reaches a failure location or is able to reach an acceptance location.

Figure 4.40 depicts an observer automaton that implements the property $synchronous_before(pt_{src}, pt_{dst})$. Actually, the observer consists of two automata: one for monitoring pt_{src} (see Fig. 4.40a) and one for monitoring pt_{dst} (see Fig. 4.40b). According to the

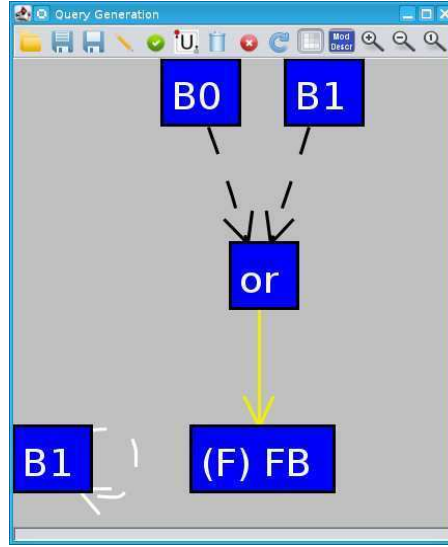


Figure 4.39: The FINSTRUCT widget that can be used to graphically design queries. On the left, the query graph for the property $\text{eventually}(a_{B_1} = 1)$ can be seen. On the right, the query graph for the property $\text{requires_strict}((a_{B_0} = 1) \vee (a_{B_1} = 1), a_{FB} = 1)$ is visible.

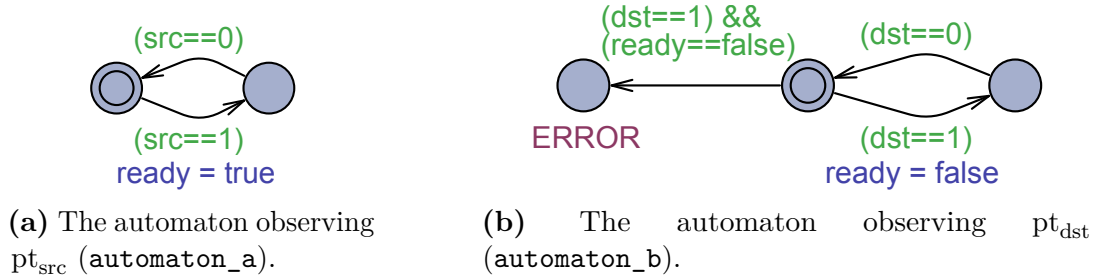


Figure 4.40: The two observer automata implementing the property $\text{synchronous_before}(pt_{src}, pt_{dst})$ (source: [Armbrust 13b]).

explanations given in Sec. 4.3.2.1, $\text{synchronous_before}(pt_{src}, pt_{dst})$ states that pt_{src} has to become true at least once before each occurrence of pt_{dst} being true. Hence, the flag `ready` is set with pt_{src} becoming true ($src==1$) by the automaton monitoring pt_{src} (automaton_a, see Fig. 4.40a). From this moment on, pt_{dst} is allowed to become true. When this happens ($dst==1$), the automaton monitoring pt_{dst} (automaton_b, see Fig. 4.40b) resets the flag. automaton_b contains a failure location named `ERROR`. The automaton transitions to this location in case pt_{dst} starts being true without a preceding occurrence of pt_{src} being true ($(dst==1) \&\& (ready==false)$). This case is indicated by `ready` not being set. As the property is fulfilled as long as automaton_b is not in the `ERROR` location, the automaton does not need an explicit acceptance location. By executing transitions in automaton_a before updating automaton_b, it is ensured that if pt_{src} and pt_{dst} start being true at the same time, `ready` is set before it is checked in automaton_b. Thus, the “synchronous” aspect of the property is correctly implemented. In order to verify the property using automaton_a and automaton_b, the two observer automata are instantiated together with the remainder of the system and it is checked whether the failure location `ERROR` can be reached using the query $E\langle\langle\text{automaton_b.ERROR}\rangle\rangle$. If this query is evaluated

to false, the **ERROR** location is unreachable and thus the system features the property **synchronous_before**(pt_{src}, pt_{dst}). If, on the contrary, the query is evaluated to true, then **ERROR** can be reached, indicating that the system does not possess the property in question.

The automatic transformation from a complex query into a combination of observer automata and a simpler query has been integrated into FINSTRUCT (see [Ropertz 12]). It can be called directly from the widget depicted in Fig. 4.39. The observer automata, the system model, and the query are sent as input to UPPAAL’s stand-alone verifier, which will execute the verification process. Its result (property satisfied/not satisfied) is directly displayed in the query graph. Thus, the proposed approach does not only meet the demand of Design Decision 9 (graphical tool support for entering queries), but also the demand of Design Decision 10 (automatic transformation of complex queries into a combination of simpler queries and observer automata). What is missing is an implementation of Design Decision 11 (visualisation of traces as changes in the behaviour network). This is described in the following.

4.3.2.4 Visualisation of Traces

The UPPAAL models created from iB2C networks according to the proposed approach can easily get large and complex. This is especially the case if observer automata are added to the original model. During the analysis of the result of the verification process, creating a trace to a witness or a counterexample is often helpful. It is not uncommon that such traces consist of numerous steps. Even with the help of UPPAAL’s simulator, keeping track of what is going on in the system model while going through the trace can be difficult. In addition, correlating the state changes in the model with changes in the original system adds further complexity.

Hence, a method for visualising traces directly in the FINSTRUCT visualisation of an iB2C network has been added to the proposed verification concept (see [Rohr 12]). A trace generated with UPPAAL based on a system model and a query can be processed by FINSTRUCT, which will calculate the corresponding changes of behaviour signals and display them in the visualisation of the original network.

Figure 4.41 depicts the visualisation of a trace generated as a witness for the property **eventually**($a_{B_1} = 1$) of the network depicted in Fig. 4.35. At first, none of the behaviours is stimulated (see Fig. 4.41a). As can be seen in Fig. 4.35, all of them should be permanently stimulated. This is achieved in the UPPAAL model with a special initialisation automaton that sends out the appropriate signals and sets the corresponding variables. In Fig. 4.41b, the initialisation automaton has completed its task—all three behaviours are stimulated. As B_1 is stimulated, it can get active (see Fig. 4.41c) and sends its activity to FB , which then also gets active (see Fig. 4.41d). The reason why the trace does not end as soon as B_1 gets active is that the system first has to reach a valid state (i.e. a state in which no **processing** flag is set).

With this visualisation of traces in the visualisation of the behaviour network in question, the demand of Design Decision 11 is also satisfied.

4.4 Discussion

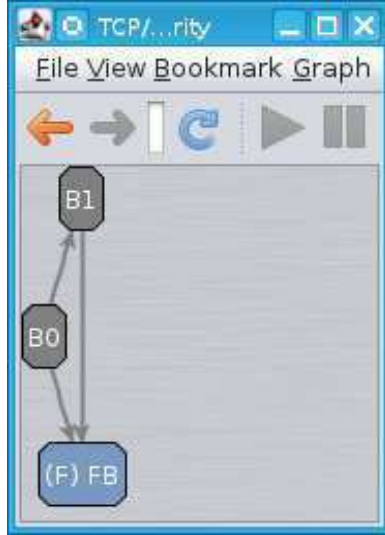
In this section, a concept for the verification of behaviour networks has been introduced. Based on the study of verification techniques presented in Sec. 4.1, model checking has

been identified as the method of choice. Due to its powerful graphical user interface, its XML-based file format as well as its stand-alone verifier, the UPPAAL toolbox has been selected as a technical basis for the realisation of the proposed concept.

With the automatic transformation of iB2C behaviour networks into networks of UPPAAL automata as described in Sec. 4.2, a technique has been developed with which a developer can easily create a representation of the system in question that can serve as a basis for model checking.

The techniques proposed in Sec. 4.3 allow for the verification of behaviour networks with sophisticated tool support. Together with the integrated visualisation of the results of the model checking, a highly automated system is available that can significantly assist a developer during the verification process. This system fulfils all necessary prerequisites for verifying iB2C networks that realise complex task sequences. In particular, this comprises the systems developed according to the concepts proposed in Chap. 3. While up to now only comparably small examples have been used to illustrate the proposed verification approach, Chap. 5 will present how an iB2C network realising a task sequence can be verified. This demonstrates the applicability of the concepts for verification introduced in this chapter to complex systems.

Figure 4.42 depicts how the second step of the verification concept (the model checking) has been incorporated into the overall concept of this doctoral thesis.



(a) At the beginning, no behaviour is stimulated.



(b) The initialisation automaton has completed its task: All behaviours are stimulated.



(c) B_1 has become active.



(d) FB has also become active as it has received the activity signal of B_1 .

Figure 4.41: Several steps of a trace that is a witness for the property $\text{eventually}(a_{B_1} = 1)$ of the behaviour network depicted in Fig. 4.35.

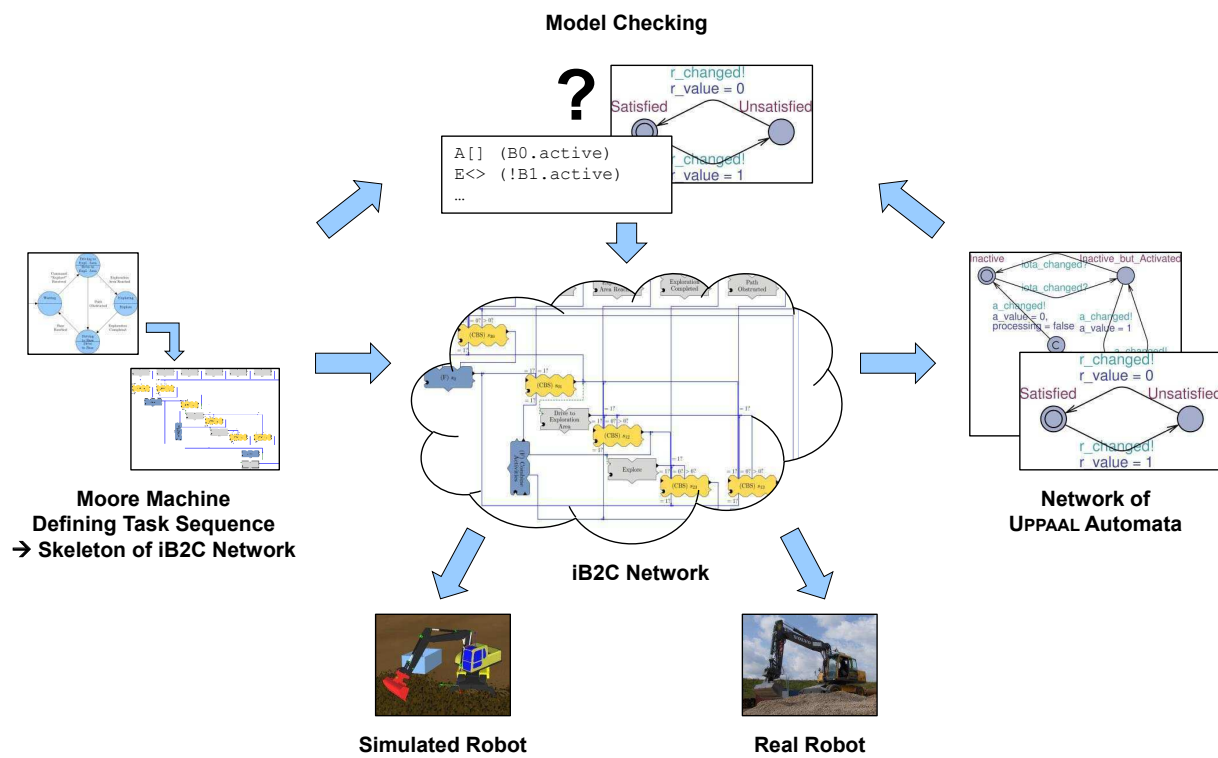


Figure 4.42: The verification of iB2C networks based on model checking has been added to the overall concept of this doctoral thesis.

5. Application Example

In this chapter, a real-world example is presented that demonstrates the application of the concepts proposed in this thesis. In Sec. 3.2.2, an updated version of the exploration example introduced in Sec. 2 has been used to illustrate the creation of an iB2C network realising a task that has been defined as a Moore machine. The example in Sec. 4.3.1 has shown how a part of the behaviour-based navigation system of the autonomous robot RAVON can be modelled as a network of UPPAAL automata in order to verify certain properties using model checking. In this chapter, by contrast, the previously introduced approaches to the design and the verification of behaviour-based systems are demonstrated using one complex application: An excavation task that is realised by a part of the behaviour-based control system of an autonomous bucket excavator.

5.1 Designing an iB2C Network Realising an Excavation Task

The example presented in this chapter originates from the control system of the autonomous bucket excavator THOR¹. The vehicle is based on a Volvo EW/180B bucket excavator (see Fig. 5.1), which is a wheeled machine that weighs approx. 18t and can produce lifting forces of around 100 kN. The long-term research goal is to develop a completely autonomous system that is able to execute typical landscaping tasks (e.g. trench excavation or slope drawing) on a previously specified target site. In [Schmidt 10], an overview of the system is given, while the authors of [Pluzhnikov 12] focus on the behaviour-based control of the arm of the excavator and the authors of [Zolynski 12] present a concept for processing point clouds that are generated by sensors installed on THOR.

In the context of this thesis, the subtasks belonging to an excavation task shall first be defined as a Moore machine, which shall then be transformed into an iB2C network following the concepts introduced in Chap. 3 (see [Armbrust 12b]).

When modelling an exploration task, first of all it has to be clarified of which subtasks the exploration consists. With regard to the first step shown in Fig. 3.13 (see Sec. 3.2.1), this

¹THOR: Terraforming Heavy Outdoor Robot



Figure 5.1: The autonomous bucket excavator THOR.

is something that should be done in cooperation with end users. In the case of a bucket excavator, the group of end users consists of skilled excavator drivers who possess extensive knowledge about the application domain. Based on discussions with them, the following five basic steps of an excavation process could be identified (cp. [Armbrust 12b]):

1. Perceiving the environment and identifying the next excavation and dumping positions based on a previously defined strategy (e.g. surface shaping or mass excavation).
2. Safely approaching the excavation position.
3. Performing a suitable excavation operation based on soil properties (e.g. grain size or material density).
4. Safely approaching the desired dumping position (possibly a dumper) without emptying the bucket.
5. Dumping the soil evenly onto the dumping position (possibly into a dumper).

This sequence of five steps is executed continuously by an excavator driver during an excavation process. According to Design Decision 1, it is defined by a Moore machine (see Fig. 5.2).

In order to execute an excavation task, THOR has to perform some additional steps. In total, this results in the following list of eight subtasks:

0. **Creating Initial Scan:** In this step, THOR creates an initial scan of its environment with its external sensors.
1. **Evaluating Scan Data:** The data of the current scan is evaluated in order to determine the next excavation position.
2. **Approaching Excavation Position:** THOR moves its bucket towards the previously determined excavation position.

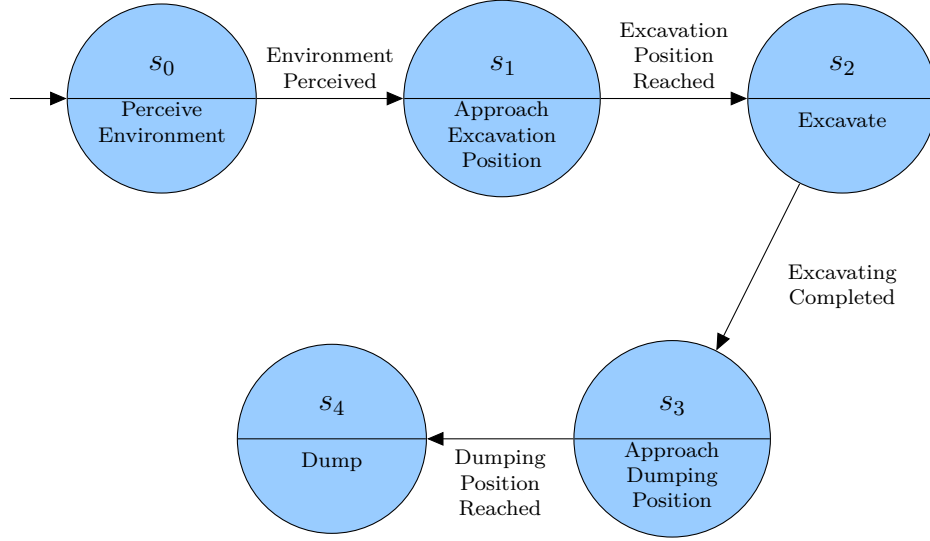


Figure 5.2: The sequence of five steps that an excavator driver repeatedly executes during a typical excavation process.

3. **Excavating:** This subtask consists of the actual arm movement with which the bucket is filled with soil.
4. **Enabling LRF and PCC:** In this step, the laser range finder (LRF) is enabled. Based on the data it provides, the point cloud collector (PCC) builds a detailed point cloud of the environment, which is used for planning the next steps.
5. **Approaching Dumping Position:** THOR moves its bucket towards the previously determined dumping position (possibly a dumper).
6. **Emptying Bucket:** THOR dumps the content of its bucket at the dumping position (possibly into a dumper).
7. **Disabling LRF and PCC:** The LRF and the PCC are disabled as one cycle of the excavation process has been completed. Furthermore, the laser scan storage (LSS) is filled with the data from the PCC.

The sequence of subtasks 1 to 7 is executed continuously until the excavation task has been completed. According to Def. 2.5, the excavation task can be defined as a Moore machine $(S, s_I, \Sigma, \Lambda, T, G)$ with the following elements:

$$\begin{aligned}
 S &= \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\} \text{ with } s_0 = \text{Creating Initial Scan,} \\
 s_1 &= \text{Evaluating Scan Data, } s_2 = \text{Approaching Excavation Position,} \\
 s_3 &= \text{Excavating, } s_4 = \text{Enabling LRF and PCC,} \\
 s_5 &= \text{Approaching Dumping Position, } s_6 = \text{Emptying Bucket,} \\
 s_7 &= \text{Disabling LRF and PCC} \\
 s_I &= s_0 = \text{Creating Initial Scan}
 \end{aligned}$$

$\Sigma = \{\text{Initial Scanning Completed, Excavation Position Found,}$
 $\text{Excavation Position Reached, Excavating Completed, LRF and PCC Enabled,}$
 $\text{Dumping Position Reached, Dumping Completed, LRF and PCC Disabled}\}$
 $\Lambda = \{\text{Create Initial Scan, Evaluate Scan Data, Approach Excavation Position,}$
 $\text{Scrape Surface, Reset and Enable PCC, Approach Dumping Position,}$
 $\text{Empty Bucket, Disable PCC and Fill LSS}\}$

$T : T(\text{Creating Initial Scan, Initial Scanning Completed})$
 $\quad = \text{Evaluating Scan Data}$
 $T(\text{Evaluating Scan Data, Excavation Position Found})$
 $\quad = \text{Approaching Excavation Position}$
 $T(\text{Approaching Excavation Position, Excavation Position Reached})$
 $\quad = \text{Excavating}$
 $T(\text{Excavating, Excavating Completed})$
 $\quad = \text{Enabling LRF and PCC}$
 $T(\text{Enabling LRF and PCC, LRF and PCC Enabled})$
 $\quad = \text{Approaching Dumping Position}$
 $T(\text{Approaching Dumping Position, Dumping Position Reached})$
 $\quad = \text{Emptying Bucket}$
 $T(\text{Emptying Bucket, Dumping Completed})$
 $\quad = \text{Disabling LRF and PCC}$
 $T(\text{Disabling LRF and PCC, LRF and PCC Disabled})$
 $\quad = \text{Evaluating Scan Data}$

$G : G(\text{Creating Initial Scan})$	$= \text{Create Initial Scan}$
$G(\text{Evaluating Scan Data})$	$= \text{Evaluate Scan Data}$
$G(\text{Approaching Excavation Position})$	$= \text{Approach Excavation Position}$
$G(\text{Excavating})$	$= \text{Scrape Surface}$
$G(\text{Enabling LRF and PCC})$	$= \text{Reset and Enable PCC}$
$G(\text{Approaching Dumping Position})$	$= \text{Approach Dumping Position}$
$G(\text{Emptying Bucket})$	$= \text{Empty Bucket}$
$G(\text{Disabling LRF and PCC})$	$= \text{Disable PCC and Fill LSS}$

The state diagram visualising this Moore machine is depicted in Fig. 5.3.

An early version of THOR's control system featured an implementation of the excavation task with a central module that realised the FSM. However, the remainder of the control system had been implemented in a strongly behaviour-based fashion, i.e. it consisted of numerous behaviours that were connected using the typical iB2C interaction types (stimulation, inhibition, fusion). Due to the realisation of the FSM in a classic way (i.e. with the logic encoded in one central component), there was a breach in the control

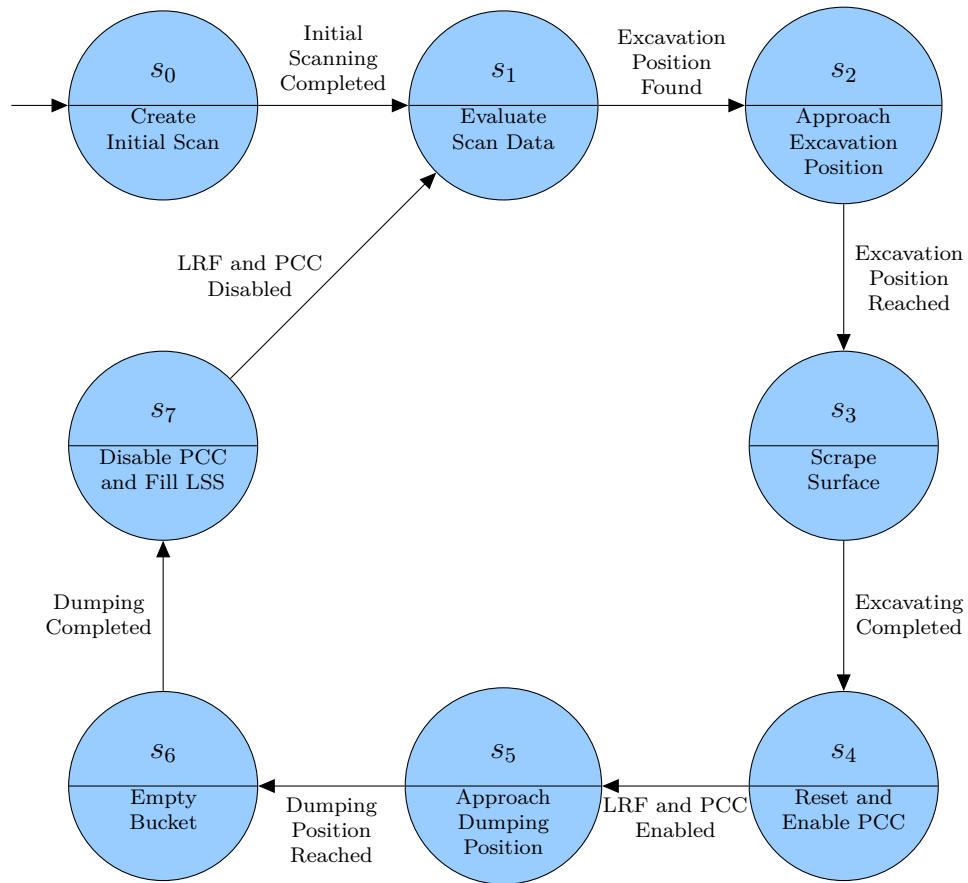


Figure 5.3: A state diagram visualising the Moore machine that represents the subtasks of the excavation task to be executed by THOR.

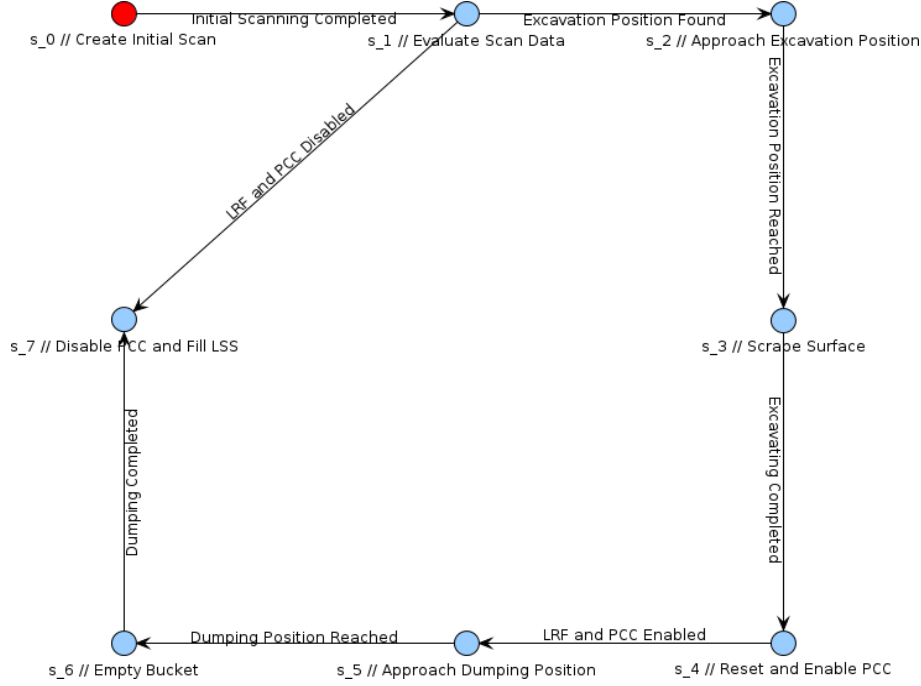


Figure 5.4: The Moore machine that represents the subtasks of the excavation task to be executed by THOR as it has been entered in the FINSTRUCT widget.

system that rendered impossible the seamless integration of the components realising the excavation task with the remainder of the network. Following Design Decision 2 (integration of concepts for realising sequences into the iB2C), the existing implementation of the excavation task in THOR's control system was supposed to be replaced with a purely behaviour-based subsystem implemented using the iB2C. With regard to Design Decision 3 (decentralised encoding of task sequences using special inter-behaviour connections), the sequential execution of the subtasks belonging to the excavation process was supposed to be encoded into the behaviour network using CBS nodes. The redesign as an iB2C network yielded an increased extensibility of the system. Furthermore, the versatile interaction features of the iB2C can be used for integrating the components that realise the excavation task with surrounding iB2C behaviours.

The construction of the behaviour network was mainly performed in line with the procedure proposed in Sec. 3.2.1: The definition of the excavation task as an FSM was done based on the experience of skilled excavator drivers, which constitute the group of the end users. As Design Decision 4 (graphical tool support for implementing complex tasks) has been realised in the form of the FINSTRUCT widget described in Sec. 3.2.4, the Moore machine depicted in Fig. 5.3 could be entered graphically by a developer without detailed knowledge about the excavator (see Fig. 5.4). Using the algorithm described in Algs. 3.1 and 3.2 (see Sec. 3.2.1), it would be possible to transfer the machine automatically into the skeleton of a corresponding iB2C network realised in FINROC. At the time of the creation of this part of THOR's control system, however, the transformation algorithm had not been implemented yet. Furthermore, the control system of THOR was still only implemented in MCA2 and not in its successor FINROC (see Chap. B). Hence, the algorithm was executed manually in order to create the skeleton of an iB2C behaviour network in MCA2. A system

specialist then added the core functionalities. The experiments were executed using the MCA2-legacy-mode of FINROC (cp. Sec. B.2).

Due to reasons of implementation, no special input behaviours monitoring conditions for transitions had to be created (cp. Line 35 of Alg. 3.1). Instead, only the activity of a behaviour executing a subtask (cp. Line 3 of Alg. 3.1) is used to determine whether a subtask is completed and the current state can be left. After the transformation, a system specialist for the bucket excavator added the core functionalities of the standard behaviours. The result of this procedure was a network of approx. 20 interconnected behaviours, which was manually extended with numerous other behaviours that are needed in several states for executing subtasks or that combine the outputs of different behaviours before they are sent downwards to the actuators of the bucket excavator. In total, the final network consists of around 90 behaviours (including the behaviours contained in behaviour groups). It is depicted in a FINSTRUCT widget in Fig. 5.5. Due to reasons of convenience, the names of some behaviours differ slightly from the names that can be derived from the Moore machine.

In the following section, it will be demonstrated how the concepts introduced in Chap. 4 can be applied in order to verify certain properties of the behaviour network realising the excavation task.

5.2 Verifying an iB2C Network Realising an Excavation Task

In this section, two applications of the previously introduced verification concepts are presented. First, it is described how predefined properties of a system can be checked. Second, it is illustrated how model checking can be used to identify properties of a (partially) unknown system (cp. [Armbrust 13b]).

5.2.1 Verifying Predefined Properties

The behaviour network realising the excavation task has been modelled following Design Decisions 5 to 8 as described in Sec. 3.1. In order to reduce the size of the state space, the target rating has not been modelled as it is not relevant here. The result of the modelling process is a network consisting of slightly over 300 synchronised UPPAAL automata.

For the verification of a system, it is necessary to know the properties that the system shall have, i.e. the properties that shall be verified during the verification process. If model checking is used, these properties have to be provided to the model checker as formulae of a temporal logic. In the context of the work on this thesis, UPPAAL is used as model checker. Therefore, the final temporal formulae have to belong to the set of formulae that UPPAAL supports (see Sec. 4.1.2.2). Due to the realisation of Design Decision 10 (automatic transformation of complex queries into simpler queries and observer automata) as described in Sec. 4.3.2, it is possible to define properties of systems that are more sophisticated than the queries directly supported by UPPAAL (see list of properties in Sec. 4.3.2.1). The implementation of the graphical tool support requested by Design Decision 9 allows for visually entering complex queries based on these properties.

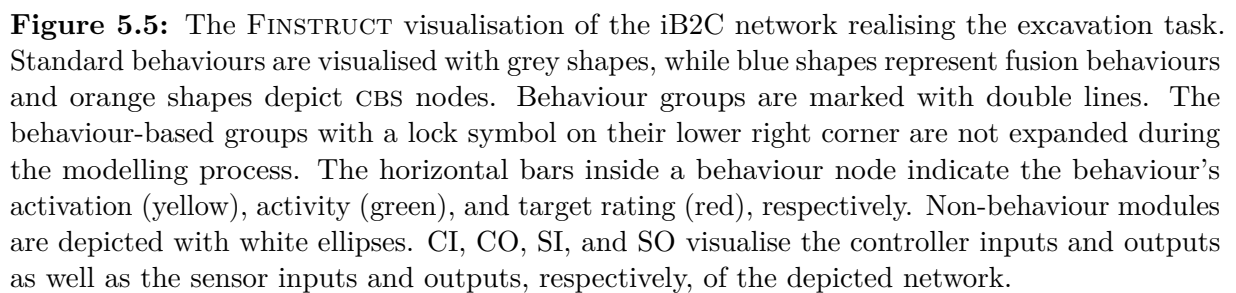




Figure 5.6: The FINSTRUCT visualisation of the query graphs defining a number of properties which request that each behaviour representing a state of the excavation process can get active. The white dashed arrows represent **eventually** properties.

The core of the behaviour network realising the excavation task has been defined using a Moore machine. While the transformation from this machine to the behaviour network was done according to an algorithm, the addition of further behaviours was done without such an algorithm. Therefore, it is possible that errors in the form of incorrect behaviour connections have been added during the manual extension of the system, resulting in a damaged core network that does not implement the Moore machine correctly anymore. Hence, a number of properties derived from the Moore machine shall be verified. As the Moore machine has been defined before the actual system has been created, the properties derived from the machine can be considered as predefined.

First of all, it is checked whether the system can reach every state of the FSM by checking whether each of the behaviours representing a state can get active. For the sake of completeness, the two CBS nodes *(CBS) Evaluate Scan Data (a)* and *(CBS) Evaluate Scan Data (b)* are also included although they do not represent a state; this is done by the fusion behaviour *(F) Evaluate Scan Data*. The following properties are used for this check:

- $\text{eventually}(a_{(CBS) \text{ Create Initial Scan}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(CBS) \text{ Evaluate Scan Data (a)}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(CBS) \text{ Evaluate Scan Data (b)}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(CBS) \text{ Approach Excavation Position}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(CBS) \text{ Excavate}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(CBS) \text{ Enable LRF and PCC}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(CBS) \text{ Approach Dumping Position}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(CBS) \text{ Empty Bucket}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(CBS) \text{ Disable LRF and PCC}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(F) \text{ Evaluate Scan Data}} = 1) \rightarrow \text{true}$

Figure 5.6 depicts the FINSTRUCT visualisation of the corresponding query graphs. Using model checking, it can be proven that all of these properties hold. As the state space when modelling the complete network depicted in Fig. 5.5 is huge, it is reasonable to use UPPAAL’s verifier with the random depth-first search order instead of the (default) breadth first search order. The help text inside UPPAAL comments that this setting is usually the best if a counterexample or witness is expected to exist. The following command was used to start the verification process:

```
$ verifyta -o 2 -S 0 -T -u Network.xml Queries.q
```

In this command, `verifyta` is UPPAAL’s stand-alone verifier, `-o 2` selects the random depth-first search order, `-S 0` disables the optimisation of the memory consumption (in order to increase the speed of the verification process), `-T` makes the verifier reuse the already explored state space in case several properties are checked, and `-u` causes `verifyta` to display a summary after the verification. Finally, `Network.xml` contains the network (including the observer automata) to be analysed and `Queries.q` stores the queries. The execution time of the above command (measured using the `time` command) was 1.246 s (user) and 0.028 s (sys) for one specific run on an AMD Opteron™ 6276 @ 2.3 GHz CPU with 256 GB of RAM running Ubuntu 12.04 LTS. 11 043 states were stored and 61 244 KiB of virtual memory were used (resident memory: 30 468 KiB). Due to the use of a *random* search order, the execution times of different runs can vary significantly. This is also mentioned in the help text inside UPPAAL. For example, another run needed 4.771 s (user) and 0.058 s (sys). The number of stored states was 63 958 and the virtual memory consumption was 99 576 KiB (resident: 70 900 KiB). A third one was not even finished after over 193 min (user), at which point `verifyta` had already used over 71 967 220 KiB of virtual memory (resident: 71 947 028 KiB) and stored 114 098 355 states.

While the verification that the behaviour network features the above properties is a technical process that might seem to have little connection to the actual application, the result of the verification is highly relevant to the real-world system. This shall be explained in the following. The core of the iB2C network has been created using an algorithm (see Algs. 3.1 and 3.2). Assuming that this algorithm is correct and that it has been implemented without introducing errors, the structure of the core network can also be assumed to be correct. However, as has already been mentioned above, the core network has been extended *manually*. In theory, this extension should not have a negative influence on the operation of the core network. In practice, however, manual additions could easily corrupt the automatically created network. For example, the connection of a behaviour’s activity output with an additional input condition port of a CBS could result in the input conditions of the CBS in question never being fulfilled. This would break the behaviour activity sequence of the CBS nodes and as a result corrupt the sequential task execution. Furthermore, during the manual work on the code, the automatically created network could easily be damaged, e.g. by accidentally removing an inter-behaviour connection. Using model checking to verify that the final BBS features the above properties, one aspect of the manually extended network could be verified: Each behaviour representing a state in the Moore machine can get active, i.e. the system can reach each of the states. In real-world terms, this means that THOR can start working on each of the subtasks.

Next, a similar set of properties is used to prove that each behaviour that executes a subtask can get active:

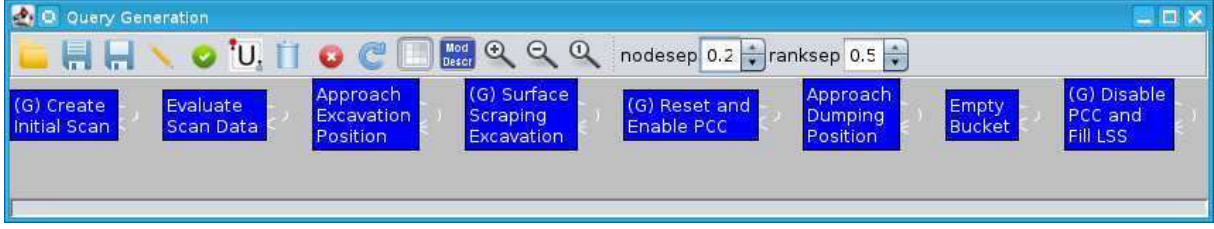


Figure 5.7: The FINSTRUCT visualisation of the query graphs defining a number of properties which request that each stimulated behaviour executing a subtask of the excavation process can get active. The white dashed arrows represent *eventually* properties.

- $\text{eventually}(a_{(G) \text{ Create Initial Scan}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{\text{Evaluate Scan Data}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{\text{Approach Excavation Position}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(G) \text{ Surface Scraping Excavation}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{(G) \text{ Reset and Enable PCC}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{\text{Approach Dumping Position}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{\text{Empty Bucket}} = 1) \rightarrow \text{true}$
- $\text{eventually}(a_{\text{Disable PCC and Fill LSS}} = 1) \rightarrow \text{true}$

The FINSTRUCT visualisation of the corresponding query graphs is depicted in Fig. 5.7. Again, as the state space of the model is huge, it is reasonable to use UPPAAL’s verifier with the random depth-first search order instead of the (default) breadth first search order. A fast run needed 6.188 s (user) and 0.048 s (sys) as well as 98 092 KiB of virtual memory (resident: 65 780 KiB) on the above-mentioned computer system. The number of stored states was 57 329. A slower run was finished after 9 min 13.278 s (user) and 2.065 s (sys) and needed 2 776 720 KiB of virtual memory (resident: 2 749 588 KiB). 3 528 194 states had to be stored.

The result of this part of the verification process is that the iB2C network can initiate the execution of each of the subtasks defined in the Moore automaton (see Fig. 5.3). Again, an error could have been introduced into the core network during the manual adaptation of the control system. This error could have resulted in one of the behaviours that execute the subtasks not being able to get active. As a consequence of this, THOR would not be able to work on all of the subtasks of the excavation process.

Another aspect relevant to the correct operation of THOR’s control system is whether the state transitions in the automaton are realised in the correct sequence. For example, before THOR starts excavating, it should have approached the excavation position. Such aspects could be realised with properties of the form

$\text{synchronous_paired_before}(a_{(CBS) \text{ Approach Excavation Position}} = 1, a_{(CBS) \text{ Excavate}} = 1),$

because the normal operation of the core network should result in the behaviours that represent states getting sequentially active and inactive again. However, behaviours added during the manual extension of the network could temporarily inhibit behaviours representing states. This could cause the above **synchronous_paired_before** property to not be fulfilled even if the system works correctly. Furthermore, the processing of such queries for the complete network needs considerable effort in terms of CPU time and memory consumption as the state space is very large. A workaround is to open the circular structure of the FSM (see Fig. 5.3) between s_7 and s_1 so that a linear sequence of states is formed. Except for the transition between s_7 and s_1 , the resulting network possesses the same properties as the original one. Instead of checking **synchronous_paired_before** properties, the following **synchronous_requires_once** properties are now verified:

- **synchronous_requires_once** $\left(a_{(\text{CBS})} \text{ Create Initial Scan} = 1, a_{(\text{CBS})} \text{ Evaluate Scan Data (a)} = 1\right) \rightarrow \text{true}$
- **synchronous_requires_once** $\left(a_{(\text{CBS})} \text{ Evaluate Scan Data (a)} = 1, a_{(\text{CBS})} \text{ Approach Excavation Position} = 1\right) \rightarrow \text{true}$
- **synchronous_requires_once** $\left(a_{(\text{CBS})} \text{ Approach Excavation Position} = 1, a_{(\text{CBS})} \text{ Excavate} = 1\right) \rightarrow \text{true}$
- **synchronous_requires_once** $\left(a_{(\text{CBS})} \text{ Excavate} = 1, a_{(\text{CBS})} \text{ Enable LRF and PCC} = 1\right) \rightarrow \text{true}$
- **synchronous_requires_once** $\left(a_{(\text{CBS})} \text{ Enable LRF and PCC} = 1, a_{(\text{CBS})} \text{ Approach Dumping Position} = 1\right) \rightarrow \text{true}$
- **synchronous_requires_once** $\left(a_{(\text{CBS})} \text{ Approach Dumping Position} = 1, a_{(\text{CBS})} \text{ Empty Bucket} = 1\right) \rightarrow \text{true}$
- **synchronous_requires_once** $\left(a_{(\text{CBS})} \text{ Empty Bucket} = 1, a_{(\text{CBS})} \text{ Disable LRF and PCC} = 1\right) \rightarrow \text{true}$

Figure 5.8 depicts the FINSTRUCT visualisation of the query graphs corresponding to these properties. Due to the large state space, the above **synchronous_requires_once** properties cannot be verified for the complete network on the available computer system. With a strongly reduced system that only consists of the behaviours realising the linear sequence of states and six additional fusion behaviours, however, it can be shown that the properties are fulfilled. The following command yielded the result in 97 h 41 min 56.967 s (user) and 3 min 29.746 s (sys):

```
$ verifyta -o 0 -S 2 -T -u Network.xml Queries.q
```

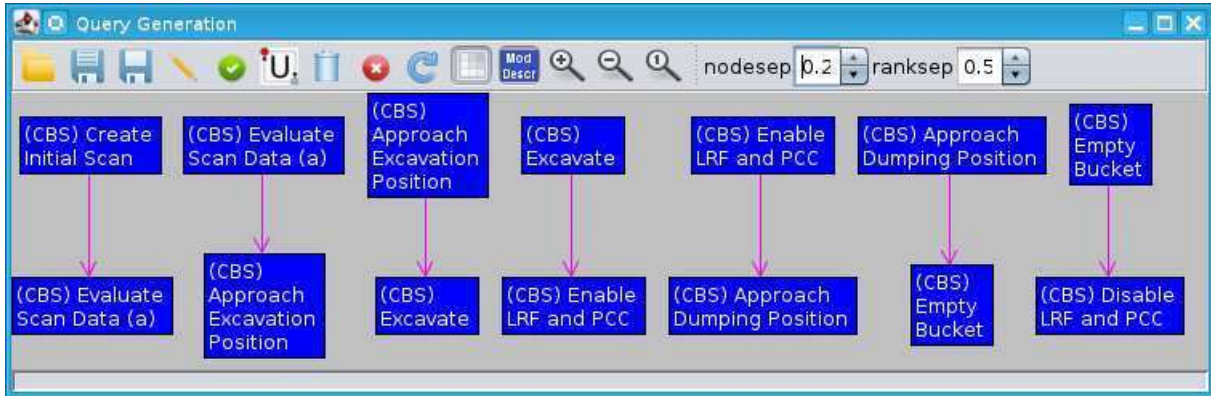


Figure 5.8: The FINSTRUCT visualisation of the query graphs defining a number of properties which request that a behaviour representing a state only gets active if the behaviour representing the previous state has already gotten active before. The magenta arrows represent `synchronous_requires_once` properties.

Instead of the random depth-first search order, the breadth-first search order (`-o 0`) has been selected here as this is recommended by the help text of UPPAAL for cases in which it can be expected that the complete state space has to be explored. 642 913 402 states had to be stored, yielding a consumption of 230 717 572 KiB of virtual memory (resident: 230 661 572 KiB).

The above properties guarantee that each CBS node only gets active when the preceding CBS in the sequence has already gotten active. Due to the manual addition of behaviours, an error could have been introduced into the system in such a way that the network does not feature the properties in question anymore. With regard to the Moore machine, the fulfilment of the above `synchronous_requires_once` properties means that a transition to a state can only occur after the automaton has been in the previous state. In terms of the excavator THOR, it means that THOR does not start working on a subtask before it has worked on the previous ones. This is important for the correct execution of the excavation task.

So far, the verification concept proposed in this thesis has been used to check whether a system possesses certain predefined properties or not. But it can also be used for another type of application, which is described in the following.

5.2.2 Analysing Partially Unknown Systems

Although there are guidelines for creating behaviour-based systems (like the ones presented in [Proetzsch 10]) and although the algorithm for transforming a Moore machine into a corresponding behaviour network that has been introduced in Sec. 3.2.1 of this thesis constitutes a formal basis for the design of a behaviour-based system, it is still possible that a developer creates a system in a more intuitive fashion. There can be different reasons for this: The available guidelines do not cover all cases, it may be impractical to implement them in certain types of behaviour-based systems, and the transformation algorithm is only applicable to tasks which can be defined as a Moore machine. Even if guidelines or an algorithm are used to design some parts of a behaviour network, it

is possible that the original network is manually modified or that further elements are added in a manual fashion. Verifying the predefined properties (see Sec. 5.2.1) after every change of the system helps in guaranteeing its correct operation. But due to the nature of behaviour-based systems, undesirable (or at least unexpected) effects can result from the interaction of behaviours. From the point of view of the original developer, the final system can be considered as (partially) unknown.

As already explained, an important question is if a behaviour can get active at all. But the developer may also be interested in the circumstances under which this can happen. Using model checking, a developer can identify these circumstances. The basic idea is to start with an existential query that yields information about whether the behaviour in question can get active at all and—if the first query is evaluated to true—to use a witness for the fulfilment of the initial query in order to identify a state in which the behaviour can get active. Once this state has been identified, it can be incorporated into an updated query in order to determine whether there are more states in which the behaviour in question can get active. By iteratively extending the query, the developer gains more knowledge about the system. This procedure shall be illustrated in the following using the behaviour network described above with the circular structure of the FSM being opened between s_7 and s_1 so that a linear sequence of states is formed. This is legitimate as the resulting network possesses the same properties as the original one except for the transition between s_7 and s_1 .

In the control system of the autonomous bucket excavator THOR, the behavioural group (G) *Approach Target Pose* is an essential component for moving the excavator's arm to the desired pose (cp. [Armbrust 13b]). This in particular comprises rotating the torso of the excavator. It is highly relevant to the safety of the system that the arm does not move unexpectedly as this could damage objects in the machine's environment or even cause harm to people. Hence, it is important to know under which circumstances this behaviour group can get active. Naturally, the overall behaviour of the core network has been defined by the FSM depicted in Fig. 5.3. But due to the above-mentioned reasons, in complex behaviour networks that have undergone some changes there may not be an up-to-date description of their functionality. For this example, it shall be assumed that it is not clear under which circumstances (G) *Approach Target Pose* can get active.

It seems reasonable to start with verifying that (G) *Approach Target Pose* can get active at all, i.e. whether the property $\text{eventually}(a_{(G) \text{ Approach Target Pose}} = 1)$ holds. In Fig. 5.9 (left), the FINSTRUCT visualisation of the corresponding query graph is depicted. A call to UPPAAL's verifier (again with the random depth-first search order) yields that the property holds and also provides a trace as witness. Design Decision 11 (tool support for displaying traces in the visualisation of the behaviour network) has been implemented as a FINSTRUCT widget, hence the trace can be visualised in FINSTRUCT.

Figure 5.10 shows the system state of the behaviour network at the end of the trace. A system state is defined by the values of the behaviour signals of all behaviours. In particular, these are the behaviours' activity values. It can be seen that (CBS) *Approach Excavation Position* is active. This results in *Approach Excavation Position* being activated. Hence, the latter can also get active, causing (F) *Approach Position* to be active. This behaviour in turn stimulates (G) *Approach Target Pose*, which can then also get active. Finally, a number of fusion behaviours gets active due to the activity of (G) *Approach*

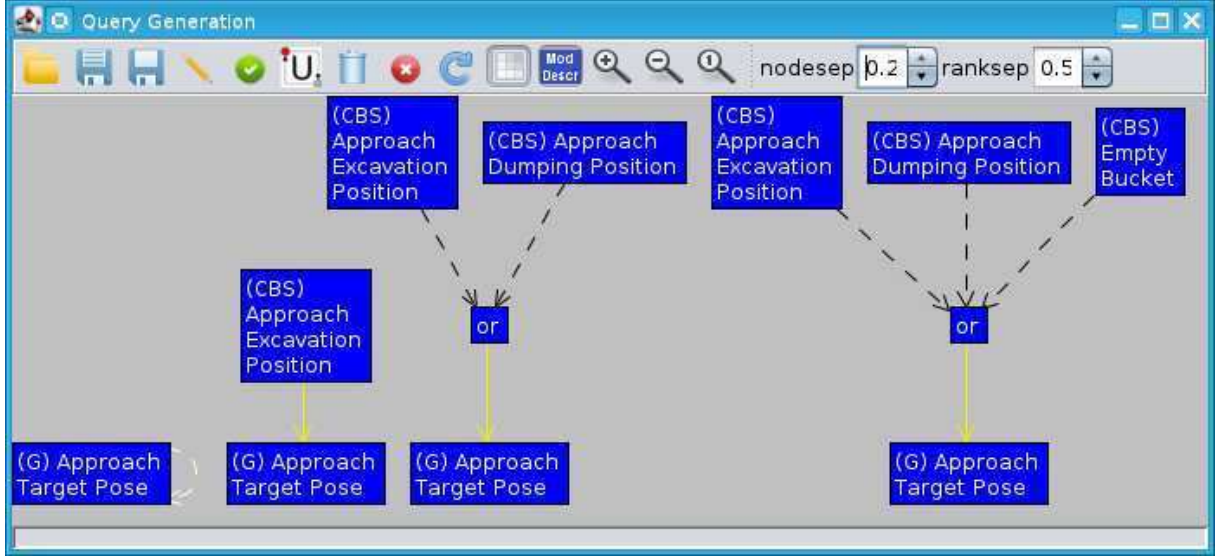


Figure 5.9: The FINSTRUCT visualisation of the query graphs defining a number of properties that help identify all states in which (G) Approach Target Pose can get active. The white dashed arrow represents an **eventually** property, while the yellow ones represent **requires_strict** properties. The black dashed arrows represent auxiliary connections for creating disjunctions.

Target Pose. From the visualisation of the trace it becomes clear that (G) Approach Target Pose can get active if (CBS) Approach Excavation Position is active, i.e. if the system is in state s_2 (see Fig. 5.3). The question is whether this is the only way in which $a_{(G)} \text{ Approach Target Pose}$ can get 1.

Another query is used to check whether $a_{(CBS)} \text{ Approach Excavation Position} = 1$ is a prerequisite for $a_{(G)} \text{ Approach Target Pose} = 1$. The corresponding property that has to be checked is the following:

$$\text{requires_strict}(a_{(CBS)} \text{ Approach Excavation Position} = 1, a_{(G)} \text{ Approach Target Pose} = 1)$$

In Fig. 5.9 (second from left), its query graph is depicted, which is transformed into two auxiliary automata and a simple query. With UPPAAL's verifier, it can be shown that the property is not fulfilled. The verifier provides a trace leading to a counterexample. Figure 5.11 depicts the last state (i.e. the counterexample) of the trace. Like in the final state of the first trace, (G) Approach Target Pose is active. However, this time the cause is (CBS) Approach Dumping Position and Approach Dumping Position being active, resulting in $a_{(F)} \text{ Approach Position} = 1$. Due to $a_{(CBS)} \text{ Approach Dumping Position} = 1$, this system state corresponds to s_5 of the Moore machine (cp. Fig. 5.3).

So far, two states have been identified in which (G) Approach Target Pose can get active. Whether there are further states can be checked using the following property:

$$\text{requires_strict}(a_{(CBS)} \text{ Approach Excavation Position} = 1 \vee a_{(CBS)} \text{ Approach Dumping Position} = 1, a_{(G)} \text{ Approach Target Pose} = 1)$$

Figure 5.9 (second from right) depicts the corresponding query graph. This property is also not fulfilled. Again, UPPAAL provides a trace to a counterexample, which can be visualised in FINSTRUCT. Its final state is depicted in Fig. 5.12. As can be seen, the

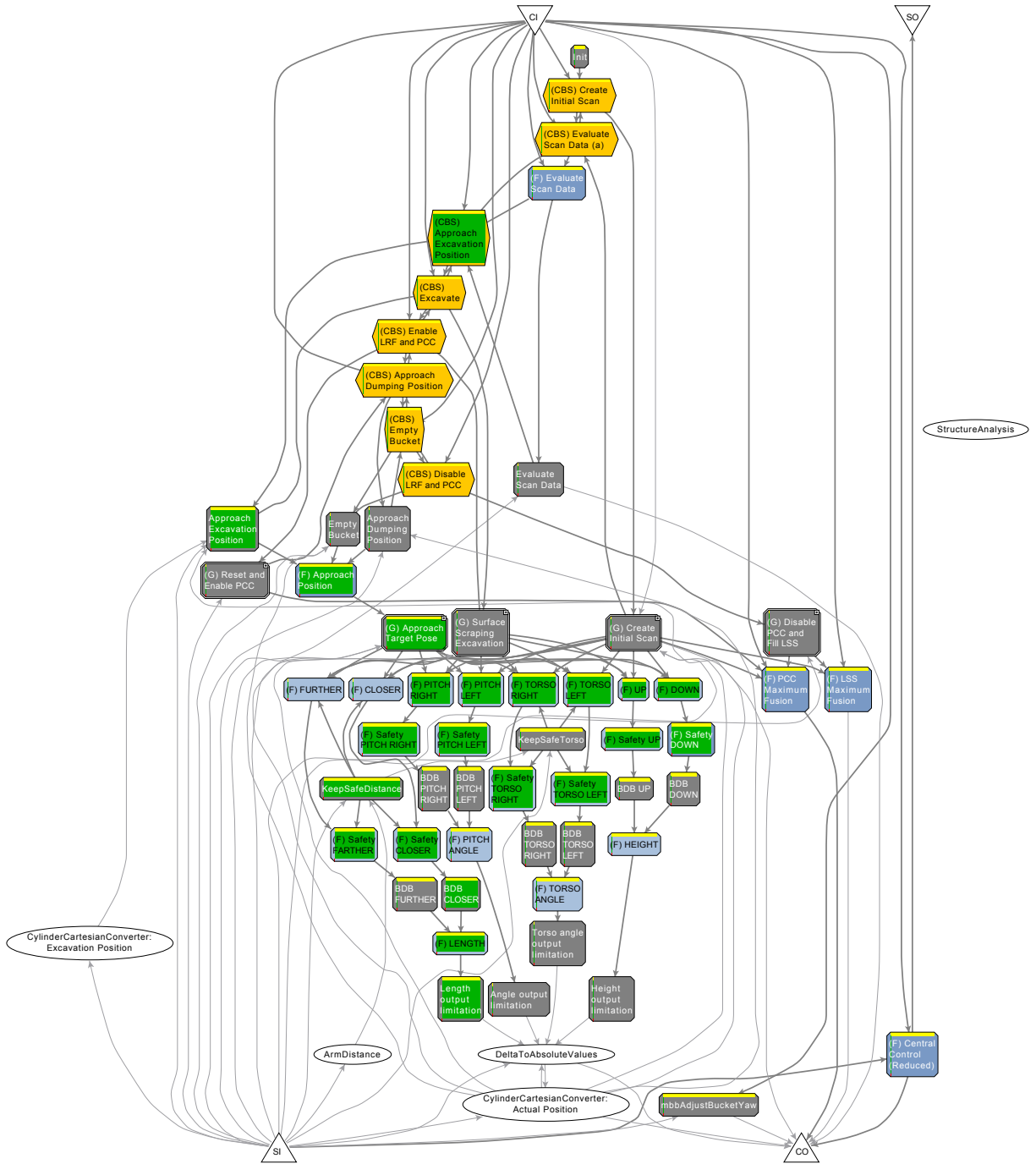


Figure 5.10: The state of the behaviour network at the end of a trace leading to $a_{(G)} \text{Approach Target Pose} = 1$. This state is a witness of the property $\text{eventually}(a_{(G)} \text{Approach Target Pose} = 1)$.

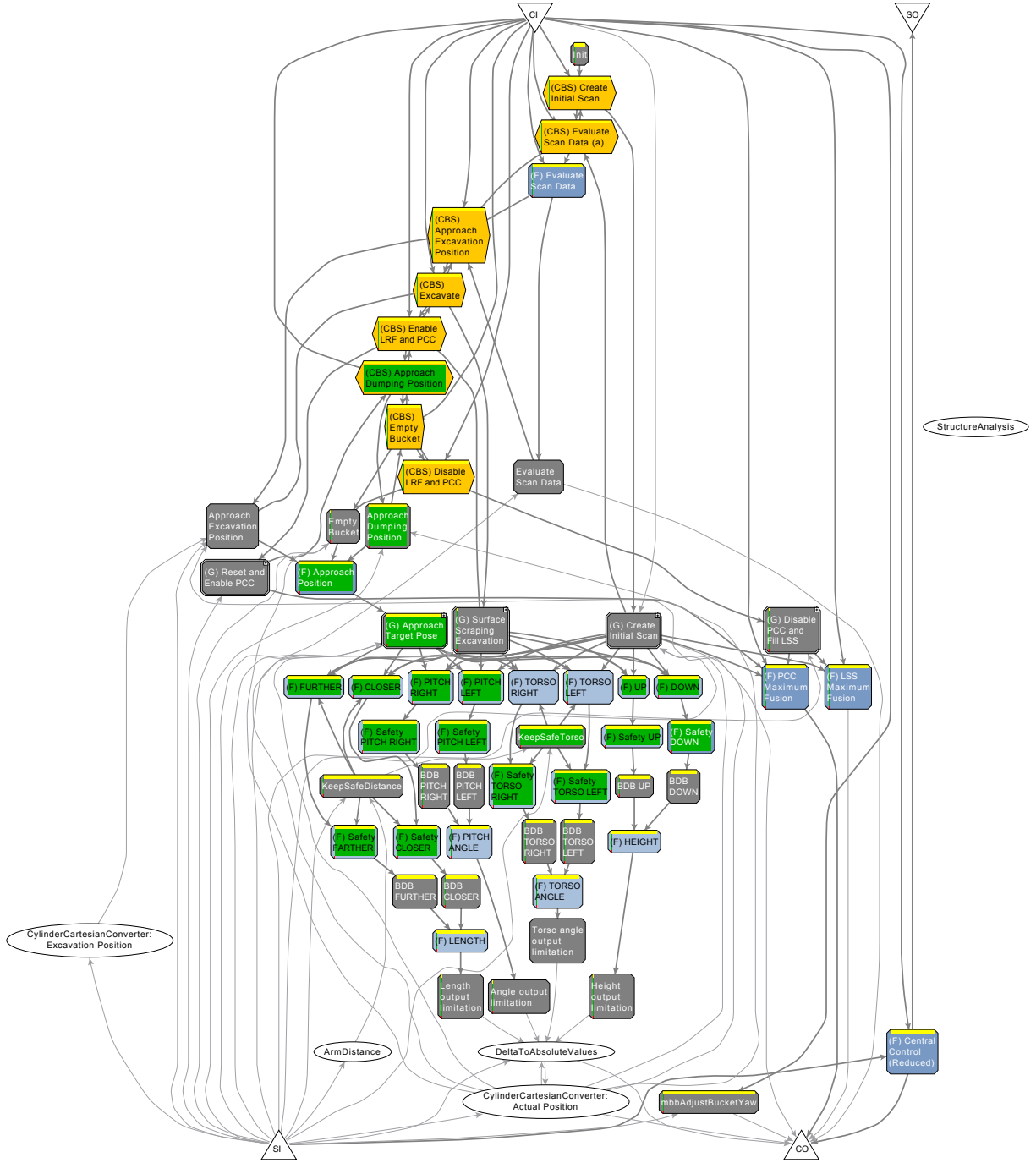


Figure 5.11: The state of the behaviour network at the end of a trace leading to $a_{(G)} \text{ Approach Target Pose} = 1$. This state is a counterexample of the property $\text{requires_strict}(a_{(CBS)} \text{ Approach Excavation Position} = 1, a_{(G)} \text{ Approach Target Pose} = 1)$.

network is in a state corresponding to s_6 of the automaton depicted in Fig. 5.3: *(CBS) Empty Bucket* is active and so are *(F) Approach Position*, resulting in *(G) Approach Target Pose* also being active.

The final step would be to check whether there is another state in which *(G) Approach Target Pose* can get active, in other words whether the system features the following property:

requires_strict $\left(a_{(CBS) \text{ Approach Excavation Position}} = 1 \vee a_{(CBS) \text{ Approach Dumping Position}} = 1 \vee a_{(CBS) \text{ Empty Bucket}} = 1, a_{(G) \text{ Approach Target Pose}} = 1\right)$. Its query graph is depicted in Fig. 5.9 (right). The verification process was started with the following command:

```
$ verifyta -o 0 -S 2 Network.xml Query.q
```

As it is expected that the query will be evaluated to true, it can be assumed that a large state space has to be explored. Hence, the breadth-first search order (`-o 0`) has been used here again. Nevertheless, the execution of the command exceeds the available RAM (256 GB) of the computation server, although the option `-S 2` (optimise memory consumption as much as possible) was used. However, this is a practical limitation and not a theoretical constraint of the presented verification approach. It can be expected that on a server with enough memory available, the query is correctly evaluated to true, indicating that *(G) Approach Target Pose* can only get active if *(CBS) Approach Excavation Position*, *(CBS) Approach Dumping Position*, or *(CBS) Empty Bucket* is active, which corresponds to the system being in one of the states s_2 , s_5 , or s_6 . In terms of the bucket excavator THOR, this means that THOR only approaches a target pose in case it is in the process of approaching an excavation position or a dumping position or in case it is in the process of emptying its bucket. While this can be expected for the core network as it has been created automatically from the specification of the task as a Moore machine (see Fig. 5.3), the manual work on the network could have introduced errors. For example, the network could have been changed accidentally so that THOR cannot approach a pose when emptying the bucket.

Besides providing information about the states in which *(G) Approach Target Pose* can get active, the above-mentioned queries have yielded an unexpected finding: If *Init*, the behaviour initiating the sequence, gets active for a second time, this can result in two CBS nodes (e.g. *(CBS) Evaluate Scan Data* and *(CBS) Empty Bucket*) being active at the same time. For most of the behaviours representing a state, inter-behaviour connections guarantee that at most one of them is active. But for the behaviours realising the initiation of the task execution sequence, there is no predecessor. Hence, the sequence can be initiated at any time. A reasonable usage of the initiation functionality of a network comprises resetting the network before re-initiating the sequence. Therefore, the undesired state in which more than one CBS is active at a time was never observed during tests. However, it cannot be guaranteed that a system is used in a reasonable way. For this reason, the network should be extended in such a way that a reset is triggered before every re-initiation.

The detection of the unexpected maloperation of the network is a typical example of the fact that testing usually cannot prove the absence of errors and that therefore, performing verification is sensible in addition to extensive testing. This emphasises the importance of the verification concepts proposed in this dissertation.

5.3 Discussion

The application example presented in this chapter has shown that the concepts introduced in Chap. 3 are sound. A complex excavation task consisting of several subtasks could be defined in a convenient way as a Moore machine and transferred into a corresponding iB2C behaviour network. This example demonstrated how a main developer and a system specialist can join forces in order to implement a complex behaviour network.

With the verification techniques introduced in Chap. 4, a developer can analyse an iB2C network. In this chapter, it has been shown how said verification techniques can be applied to a real-world system that realises a task sequence. Several predefined properties could be checked and it could also be shown that the proposed methodology allows for determining properties of (partially) unknown systems. Furthermore, an undesired system behaviour that could occur due to unreasonable usage could be identified.

But the experiments have also revealed a practical limitation of the proposed verification technique: The state space that has to be explored during a verification process gets large, resulting in a high memory usage and a long processing time. For larger networks and more complex queries, even 256 GB of RAM are insufficient. There are basically two ways of approaching this problem:

1. Reduce the size of the overall UPPAAL model by reducing the size of the templates and by improving the modelling algorithm.
2. Reduce the UPPAAL model to that part which is needed for checking the query in question.

Reducing the size of the templates will directly result in a smaller size of the overall model as the latter is built up of instances of the templates. Especially the model of the CBS is comparably complex and might offer some room for improvement. Furthermore, improving the algorithm that creates the overall system model could also help in reducing the total size. A large number of locations are only a technical workaround as UPPAAL does not support more than one synchronisation per edge. With a more sophisticated modelling algorithm, it could be possible to eliminate some or all of these intermediate locations.

A completely different approach to solving the problem of a huge state space is to reduce the model that is used for the model checking process to that part of the overall model that is actually needed. In a way, this is already done by not expanding some behaviour groups, but instead modelling them as standard behaviours. However, the identification of parts of the network that are not relevant for the verification in question could be done in a more sophisticated way. Naturally, this is not a trivial process as the interconnections of the behaviours contained in the network have to be analysed.

Both ways of reducing the state space are beyond the scope of this work. But even without such improvements, the verification concept proposed in this thesis is applicable to real-world systems as this chapter has shown.

6. Conclusion

The ability to execute sequences of tasks is essential for autonomous vehicles working in complex environments. Hence, it is essential that their control systems provide the necessary support and that the correct operation of their control systems can be guaranteed. The work described in this dissertation has dealt with the implementation of task sequences in behaviour-based systems. Its scientific contribution is an integrated concept for the design and verification of behaviour-based systems realising task sequences. This concept is illustrated in Fig. 6.1. It is based on the following essential achievements:

1. A behaviour-based architecture has been equipped with the ability to realise complex tasks consisting of sequences of subtasks.
2. An algorithm has been developed that transforms a formal description of a task into the skeleton of a behaviour network.
3. A concept for modelling a behaviour network as a network of automata for verification purposes has been formulated.
4. An approach to performing formal verification based on these networks of automata has been designed.

In the remainder of this chapter, the main results of the preceding chapters are recapitulated and the proposed concepts are evaluated. Finally, an outlook on future work is given.

6.1 Summary

Two objectives have been the starting point of this dissertation: the development of a technique for realising task sequences in behaviour networks (cp. Objective 1) and the development of a technique for verifying behaviour networks that realise task sequences (cp. Objective 2). On the basis of these two objectives, the following results have been generated:

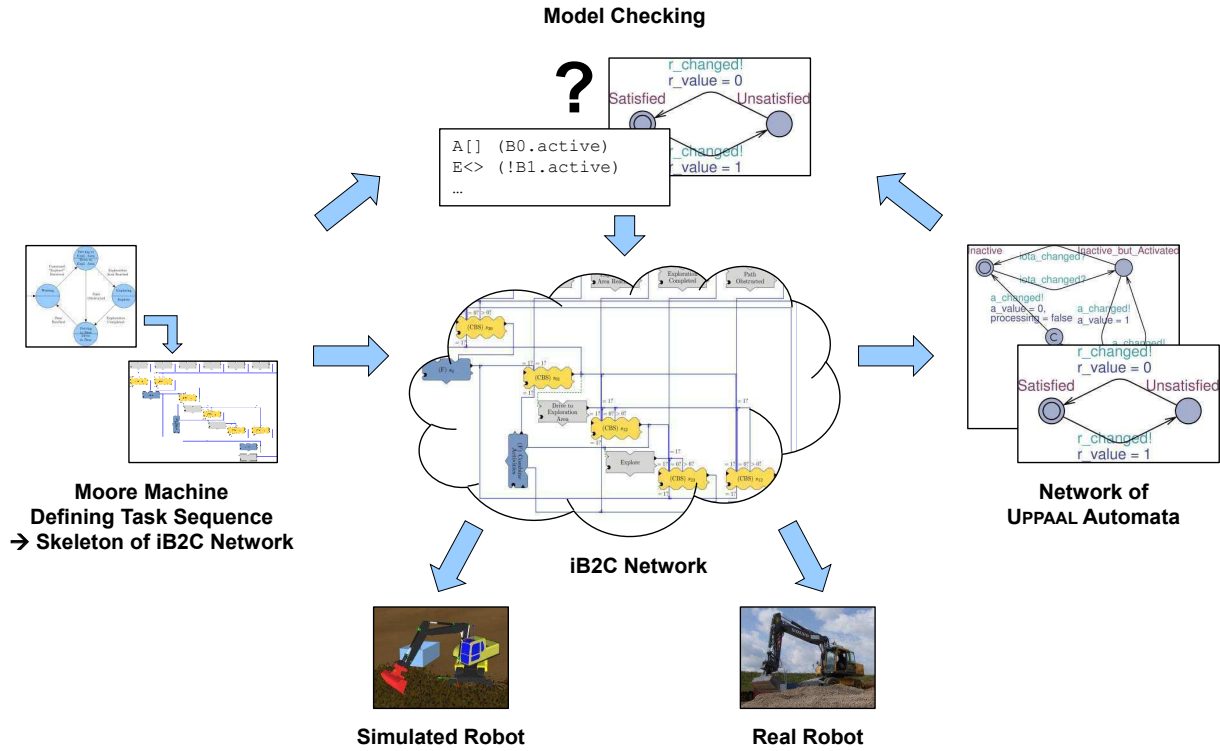


Figure 6.1: The overall concept of this doctoral thesis: The task of a behaviour network is specified as a Moore machine, which is transformed into the skeleton of a behaviour network. With the addition of the core functionalities, the final network is created. This can be used to control a simulated or a real robot. For verification purposes, a model based on FSMs is created, which is passed to a model checker along with queries that are derived from the specification of the system. The results of the verification process are used to improve the network.

As complex tasks consisting of sequences of subtasks are the main topic of this dissertation, different ways of defining sequences have been described in Chap. 2. Finite-state machines (in particular Moore machines) have been selected as the notation of choice for the work described in this dissertation because on the one hand they allow for a graphic description of complex tasks, while on the other hand they are formal enough to be processed automatically (cp. Design Decision 1).

In addition, a number of techniques for the realisation of sequences in robot control systems have been presented in Chap. 2. The options of how to realise sequences in a robot control system heavily depend on the architecture of the system. For this reason, the choice of the control architecture has been crucial for the work described in this thesis. A class of architectures that have a number of advantages over others are the behaviour-based architectures: The distribution of the overall functionality over several entities allows for parallelising the development, implementation, and testing of the system. Furthermore, it fosters the reuse of components across different platforms and the redundant implementation of critical functionalities in different modules. Due to their advantages, behaviour-based architectures have been selected as the type of choice for realising task sequences. Therefore, the developed concepts have been integrated into the behaviour architecture iB2C (cp. Design Decision 2). With respect to the distribution of functionality over a number of behaviours in a behaviour-based system, the decentralised encoding of sequences into behaviour networks using special inter-behaviour connections has been chosen as implementation technique for sequences in the context of the work described in this thesis (cp. Design Decision 3). Furthermore, it has been stated that graphical tool support for the creation of such networks should be available in order to facilitate a developer's job (cp. Design Decision 4).

In Chap. 3, it has been demonstrated how behaviour activity sequences can be realised in the behaviour architecture iB2C. With regard to the findings of Chap. 2, the sequences in question are directly encoded into behaviour networks in a decentralised fashion using special inter-behaviour connections. For this purpose, a local coordination behaviour has been integrated into the iB2C. Its activity depends on the fulfilment of different types of conditions which in turn depend on the activities or target ratings of connected behaviours. It has been shown that by using this local coordination behaviour, the sequential execution of tasks can be realised. Furthermore, behaviour networks can detect the sequential perception of structures in a robot's environment, which allows for assessing if the robot has encountered a specific situation. The development of these techniques has been done in accordance with Design Decisions 2 and 3.

Chapter 3 has also presented a three-step concept for creating an iB2C network that executes a complex task. According to this concept, a main developer defines the task as a Moore machine (cp. Design Decision 1) together with end users, who possess knowledge in the area of application of the system. This can be done graphically using a special widget. With an algorithm that has been developed in the context of the work described in this thesis, the Moore machine can be automatically transformed into the skeleton of a behaviour network. Core functionalities can then be added by system specialists, i.e. people who are familiar with the capabilities of the robotic platform. Using the example of an exploration robot, the soundness of the transformation algorithm has been demonstrated. The demand of Design Decision 4 for graphical tool support has been met by the design of a graphical user interface for defining Moore machines.

Behaviour networks realising complex tasks tend to be complex themselves. In many cases, at least parts of them have been designed manually, possibly without guidelines. The result are systems with (partially) unknown functionalities. In order to determine whether a given behaviour-based system conforms to its specification, formal verification techniques can be applied. This has been the starting point of the work described in Chap. 4. First of all, an introduction to the topic of verification has been given. This has comprised the presentation of two common verification techniques: deductive reasoning and model checking. Model checking can be highly automatised and is able to generate witnesses or counterexamples for queries. Furthermore, a powerful, easily usable model checking toolbox (UPPAAL) is available. Therefore, model checking with UPPAAL has been selected as a verification technique for the work described here.

When using model checking based on UPPAAL for verifying an iB2C network, the network has to be modelled as a network of UPPAAL automata. Several fundamental decisions about the nature of the model have been made. They have been justified in Chap. 4. Due to the importance of the behaviour signals for iB2C systems, it has been decided that these signals and their calculation shall be represented in the model (cp. Design Decision 5). In order to reduce the state space, their value range has been limited to $\{0, 1\}$ (cp. Design Decision 6). Furthermore, it has been decided to model each behaviour as a fixed set of interconnected UPPAAL automata that depends on the type of the behaviour (cp. Design Decision 7). This is a compromise between numerous small automata with a lot of synchronisation overhead and few very large automata with the advantage of a less complex synchronisation. In order to maximise the usability, it has been decided that the proposed concepts should be implemented with a high degree of automation (cp. Design Decision 8).

The resulting UPPAAL models of standard and fusion behaviours as well as CBS nodes have been presented in detail in Chap. 4. Furthermore, the modelling algorithm has been described and information about the complexity of the resulting models in terms of the numbers of locations and edges has been provided. Based on the queries supported by UPPAAL, it has been shown how properties of behaviour networks can be checked. An example based on the navigation system of the autonomous off-road robot RAVON has been used to illustrate the verification process.

The observation that the manual creation of complex queries is error-prone and the general limitations on the queries that can be processed by UPPAAL have led to the insights that a graphical tool should assist the developer during the verification process (cp. Design Decision 9) and that tool support should be available for transforming complex queries into a combination of observer automata and simpler queries that can be processed by UPPAAL (cp. Design Decision 10). These requirements are fulfilled by the presented widget with which the developer can graphically design queries, which are then automatically transferred so that they are accepted by UPPAAL's verifier as input.

Finally, it became apparent that the visualisation of traces in UPPAAL's simulator is hard to read and that it is tedious to draw conclusions about the original system from a list of changes in the automata. Hence, it has been stated that the developer should be able to track changes in a network of UPPAAL automata in the form of the resulting changes in the corresponding iB2C behaviour network (cp. Design Decision 11). How such a functionality can be implemented as a widget has also been described in Chap. 4.

In Chap. 5, all of the concepts introduced before have been applied to a real-world example. First, an excavation task to be realised by the autonomous bucket excavator THOR has been designed as a Moore machine, which has then been transformed into the skeleton of an iB2C network according to the transformation algorithm presented in Chap. 3. The actual functionality has been added by a system specialist. Second, the resulting behaviour network has been modelled as a network of synchronised UPPAAL automata according to the concept presented in Chap. 4. With the help of the presented tool support, several complex queries have been used to verify parts of the behaviour network and the resulting traces have been displayed in the visualisation of the network.

6.2 Evaluation

The examples of Secs. 3.1.2.1 (turning manoeuvres) and 3.1.2.2 (dead end detection) have shown that the proposed concept for encoding task sequences into behaviour networks in a decentralised fashion using a local coordination node is sound. The two examples are part of the control system of an autonomous mobile robot. This proves the applicability of the concept to real-world systems.

The weak point of this approach—the necessity to manually design large behaviour networks—has been overcome by the development of graphical tool support based on an algorithm transforming Moore machines into corresponding skeletons of behaviour networks. The application of the concept to the control system of an autonomous bucket excavator (see Sec. 5.1) has demonstrated that the three-step workflow proposed in Sec. 3.2.1 can be applied to complex, real-world applications.

With the example of Sec. 4.3.1, which has verified that a part of the behaviour-based navigation system of an autonomous mobile robot features certain properties, the applicability of the presented verification concept to behaviour networks of real control systems has been demonstrated. Even with a set of comparably simple queries, it could be shown that the control system fulfils a number of requirements.

During the described verification process, it became apparent that the manual creation of queries is tedious and error-prone. Furthermore, the restrictions imposed by UPPAAL on the queries hindered the applicability of the approach to more complex verification tasks. Both challenges have been met by introducing powerful tool support that greatly facilitates the input of queries and overcomes the aforementioned limitations caused by UPPAAL. This has been demonstrated by using again the control system of the above-mentioned autonomous bucket excavator (see Sec. 5.2).

An issue that remains is the large amount of memory consumed by the verifier during more complex verification processes. This issue had a negative effect on the experiments described in Sec. 5.2 and can severely complicate or even render impossible the formal analysis of larger networks. Hence, it should be the primary target of improvement measures. Two starting points have already been mentioned in Sec. 5.3. In the following section, these measures will be discussed once again.

6.3 Outlook on Future Work

A possible starting point for future work concerns the modelling of behaviour-based systems. At the moment, the widget for defining Moore machines does not support the creation of

hierarchical state machines. In complex systems like the control system of THOR, however, structures of hierarchical FSMs can easily occur. Therefore, the expansion of the widget and the corresponding transformation algorithm by support for hierarchical FSMs seems beneficial.

Although the presented approach facilitates the creation of behaviour networks that execute sequences of tasks, it does not specify how these networks should be integrated with other behaviour-based components. In the current state, the method in which the two sub-networks are integrated is not specified, i.e. it is up to the developer to decide on how to connect the involved behaviours. Some guidance is provided by the already mentioned principles proposed in [Proetzsch 10] (see Sec. 2.2.1). However, there are currently no special guidelines targeting the integration of said networks. In the context of future work, principles for this case could be formulated. Furthermore, the available widget for defining Moore machines could be extended in a way in which the developer could already in this step specify aspects of the network integration. These aspects could then be processed by an algorithm during the creation of the behaviour network so that there is less manual work for the developer.

Two options for improving the usability of the presented verification approach are related to the problem of the huge memory consumption. They have already been mentioned in Sec. 5.3. First, the size of the complete UPPAAL model could be reduced by reducing the size of the UPPAAL templates. In particular, the elimination of the intermediate committed locations would have a strong positive impact. Second, reducing the model that is used for the verification process to the exact part that is actually needed could tremendously reduce the memory consumption of the process. To a certain degree, this is already done by modelling a behaviour group as a standard behaviour in case the behaviours contained in the group are not of relevance to the verification process. But the selection of such groups is a manual process. An algorithm that automatically identifies the behaviours that have to be modelled for verifying a certain property and which ignores the remaining behaviours during the creation of the UPPAAL model would constitute a major improvement of the current (manual) selection of the behaviours to be modelled. In addition, it would allow for applying the proposed verification concept to more complex systems.

While the verification concept proposed in this dissertation targets *software* failures, it can be extended to also model the occurrence of *hardware* failures. In [Kiebusch 14], an approach to incorporating sensor failures during the verification process is presented. For this purpose, the modelling algorithm has been extended so that it can identify so-called safety behaviours that deal with a robot's collision avoidance. Whenever the algorithm encounters a safety behaviour, it creates instances of special templates for modelling a safety behaviour's activity and target rating. Furthermore, the algorithm creates instances of special UPPAAL templates that model sensors. Using the final UPPAAL model, it is possible to check under which conditions a safety behaviour gets active, i.e. under which conditions the robot's anti-collision system takes action. Hence, the work described in [Kiebusch 14] shows that the verification concept presented in this dissertation has the potential to be extended to further applications.

Appendices

A. Robot Control Architectures

When talking about architectures in robotics, the question arises what is meant by the term “architecture”. Looking at computer science literature yields various definitions for the term. IEEE 1471 (see [IEEE 1471 00]), for example, states that an architecture “is defined by the recommended practice as the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution”. The standard has meanwhile been superseded by ISO/IEC/IEEE 42010 (see [ISO 42010 11]), which defines an architecture as the “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution”. Adapting the ISO/IEC/IEEE 42010 definition to robotics yields the following definition:

Definition A.1: Robot Control Architecture

A *robot control architecture* (or simply architecture) consists of the fundamental concepts or properties of a robot control system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.

Based on the remarks about robot control in [Matarić 08], the term “robot control system” can be defined as follows:

Definition A.2: Robot Control System

A *robot control system* is a system that takes information about a robot’s environment through the robot’s sensors, processes it as necessary in order to make decisions about how to act, and executes actions in the environment using the robot’s actuators.

There are two opposing approaches to designing the architecture of a robot control system: the deliberative and the reactive approach. Furthermore, in hybrid architectures, a layer that has been realised following the deliberative approach is combined with a layer that has been realised following the reactive one. In the following, the different approaches will be presented and examples will be given.

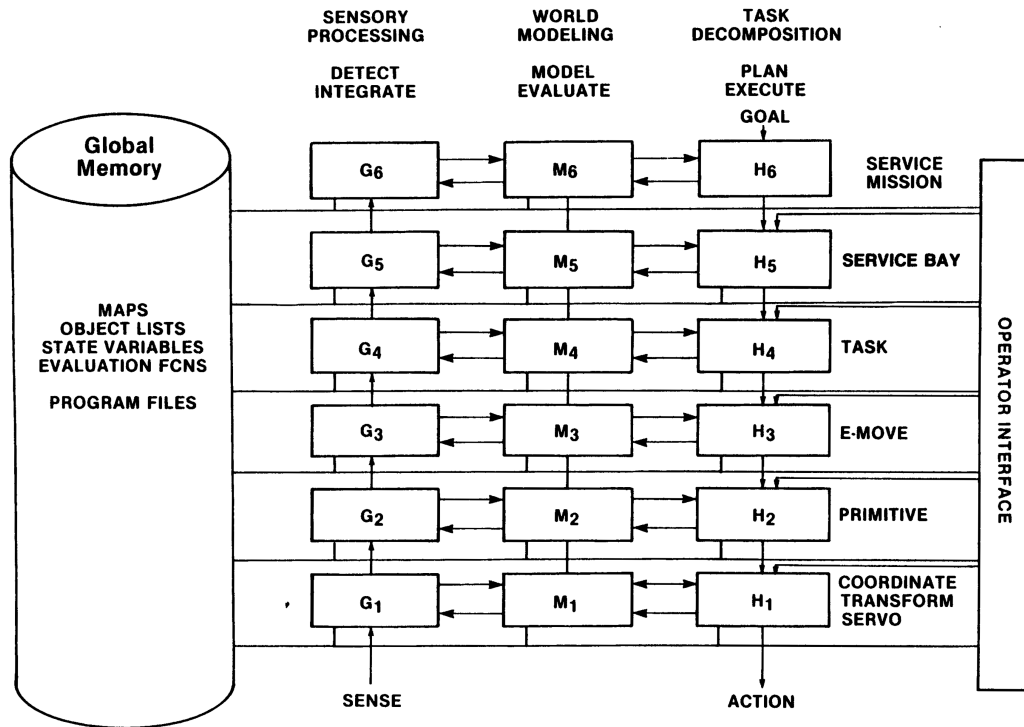


Figure A.1: Overview of the components of NASREM (source: [Albus 89a]).

A.1 Deliberative Robot Control Architectures

Deliberative robot control architectures focus on reasoning about which actions the robot should execute next. This is done based on all available sensor data and an internal representation of the robot's environment. Robot control systems designed in a deliberative fashion typically work according to the sense-plan-act (SPA) principle, which means that they take environmental information gathered by the robot's sensors, integrate it into a large world model, perform reasoning based on this world model in order to figure out what to do next, and then carry out the resulting plan. Due to the dependence on the world model, this control approach is also known as sense-model-plan-act (SMPA).

In the 1980s, research at the USA's National Bureau of Standards (NBS), which was later turned into the National Institute of Standards and Technology (NIST), led to the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) (see [Albus 87] and [Albus 89b]). While its predecessors were used in laboratory robotics and autonomous undersea vehicles, amongst others (see [Albus 89a]), NASREM was developed for the control system of a space station. NASREM is a prominent reference architecture for the SMPA approach.

Figure A.1 depicts the structure of NASREM. The system is built up of three pillars (sensory processing, world modelling, and task decomposition). Each of the pillars is separated hierarchically into different modules. What is noteworthy here is that modules only communicate with the modules of different pillars but on the same layer, and with the modules on the next higher or next lower layers but in the same pillar, i.e. the model does not allow directly connecting arbitrary components.

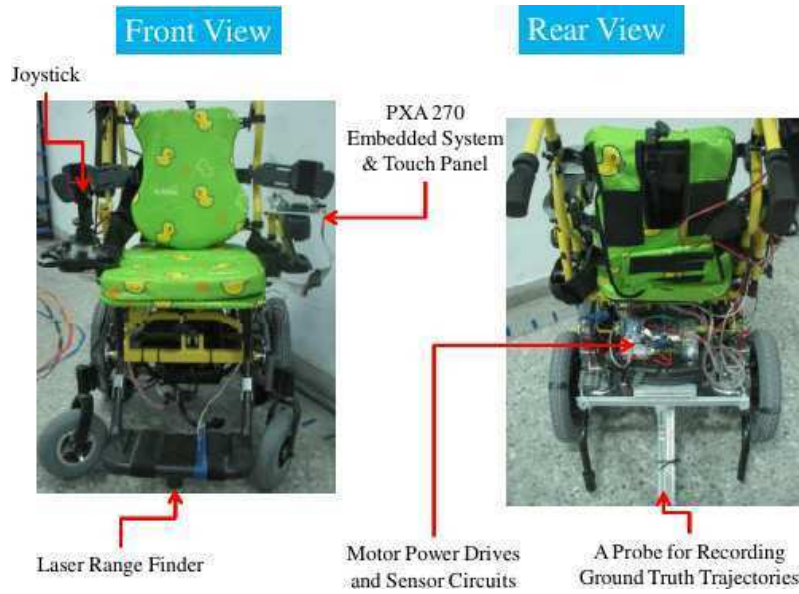


Figure A.2: The robotic wheelchair prototype (source: [Kuo 11]).

The strength of deliberative architectures lies in their ability to reason about the current state of the robot's environment and possible changes in the future. Furthermore, they can deal with large amounts of data and apply complex planning algorithms for figuring out the robot's next actions. The downside is that the reasoning and planning processes need a considerable amount of time, thus complicating or even rendering impossible the fast reaction to changes in the environment. Besides, integrating a huge amount of data into a single world model can easily create a bottleneck for data transmission and necessitates solving contradictions in the sensor data before the integration into the model is possible.

A.2 Reactive Robot Control Architectures

The essence of reactive robot control architectures is a tight coupling of sensors and actors, often using simple control algorithms. In contrast to deliberative architectures, reactive ones do not possess a world model into which all sensor data is integrated and on which planning algorithms are executed. Instead, the data originating from the robot's sensors is directly processed by (typically) simple algorithms and the results are directly used to control the robot's actuators. This allows for very fast reactions to changes in the environment. However, such control systems are sensitive to temporary sensor failures and lack the ability to predict what will happen in the future. As they are not capable of creating complex plans, they are not suited for realising sophisticated tasks and are thus often combined with a deliberative component into a hybrid robot control architecture (see Section A.3).

A current example for a purely reactive control system is given in [Kuo 11]. Its authors describe a robotic wheelchair that assists its user by providing collision avoidance capabilities. The wheelchair is equipped with a laser range finder for obstacle detection, wheel encoders for odometry-based localisation, an embedded platform featuring a touch panel, and a joystick (see Fig. A.2). The detection area of the laser range finder is split up into

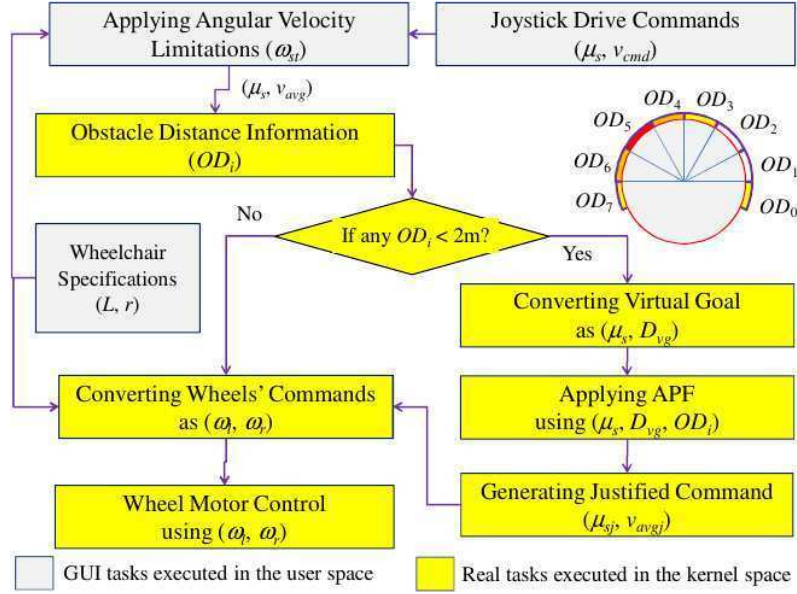


Figure A.3: Schematic of the navigation system of the robotic wheelchair (source: [Kuo 11]).

eight equally sized polar sectors. A schematic of the navigation approach is depicted in Fig. A.3. As can be seen, the user's drive commands μ_s and v_{cmd} (calculated from the joystick position) are converted into linear wheel speeds ω_l and ω_r in case there is no obstacle nearby (obstacle distance $OD_i \geq 2\text{ m } \forall i$). If there is an obstacle, justified commands μ_{sj} and v_{avgj} are generated based on a virtual goal and an artificial potential field (APF) approach.

In APF approaches, the target location represents an artificial attractive potential that results in a force \vec{F}_{att} attracting the robot towards it. Obstacles, by contrast, represent artificial repulsive potentials that result in forces $\vec{F}_{rep,i}$ pushing the robot away from them. From these two types of forces acting on the robot, the resulting force \vec{F}_{res} can be calculated, which determines the robot's speed and angular velocity. The navigation approach presented in [Kuo 11] does not feature any goal-driven navigation. Hence, a virtual goal is created based on μ_s and the average wheel speed v_{avg} with its distance D_{vg} to the robotic wheelchair being proportional to v_{avg} . The repulsive forces $\vec{F}_{rep,i}$ are calculated from the distances to the obstacles OD_i and combined to \vec{F}_{rep} :

$$\vec{F}_{rep} = \sum_{i=1}^6 \vec{F}_{rep,i}$$

From the attractive and repulsive forces, the resulting force \vec{F}_{res} is determined. Furthermore, the maximum magnitude of the attractive force F_{Matt} is calculated as $F_{Matt} = k_a \cdot D_{far}$, with k_a being an attraction constant and D_{far} being the goal distance at which the wheelchair shall drive with its maximum linear velocity v_{max} . The justified commands v_{avgj} and μ_{sj} are then determined using the following two formulae, where θ_{res} is the angle of \vec{F}_{res} with respect to the joystick coordinate system:

$$v_{\text{avgj}} = \frac{\sqrt{\vec{F}_{\text{res}}^T \cdot \vec{F}_{\text{res}}}}{F_{\text{Matt}}} \cdot v_{\text{max}} \quad \mu_{\text{sj}} = \cos \theta_{\text{res}}$$

This type of navigation approach is typical of reactive systems: It features straightforward calculations and goes without storing state information. However, in theory it also exhibits a typical problem of those systems: APF approaches can suffer from local optima problems and thus the robotic wheelchair could be caught between obstacles without being able to find a way out. The authors of [Kuo 11] are aware of this fact and argue that in practice, this problem is solved by the user commanding the wheelchair in a proper way. While this is true, the authors' work thus heavily relies on the user dealing with high-level navigation. Therefore, this reactive approach would not be sufficient for fully autonomous systems without a human in the loop. Some high-level, deliberative component would have to be added.

A.3 Hybrid Robot Control Architectures

The developers of purely reactive architectures and those of purely deliberative ones follow contrasting approaches. While the deliberative approach is especially suited for realising planning on a high level, it fails when it comes to fast robot reactions to changes in the environment. Reactive components, on the contrary, are often the first choice for realising collision avoidance, but lack the ability to reason about the world around the robot and create plans.

This led to the development of hybrid architectures, in which the low-level robot control is governed by reactive components, while a deliberative system deals with high-level task control. In many cases, an intermediate layer builds the interface between the two. Typical tasks of this layer include dividing complex tasks provided by the deliberative layer into smaller subtasks, enabling or disabling components of the lowest layer to realise sequences of process execution, preprocessing sensor information from the lowest layer and passing the result on to the highest layer, and asking the deliberative layer for advice in case the reactive one cannot solve a problem on its own. Therefore, this type of robot control architectures is often referred to as three-layer architectures (see [Gat 98] for an overview of such architectures).

The middle layer is crucial to the correct operation of the system. As the authors of [Hexmoor 95] point out, the “middle layer is the key” that serves “as a mediator between deliberation and reactivity”. Typically, the time scale of operations executed on the different layers increases from the bottom to the top. This is an aspect that can also be found in humans. [Newell 90], for example, describes different time scales of human action.

One of the first researchers who integrated deliberative and reactive components in one control architecture was James Firby. Figure A.4 depicts the three-layered architecture he envisaged in [Firby 89]. A planning layer translates high-level goals into sequences of tasks and inserts them into the task agenda. Its middle layer consists of the RAP¹ system. A RAP is a component that encapsulates all information necessary for executing a certain

¹RAP: Reactive Action Package

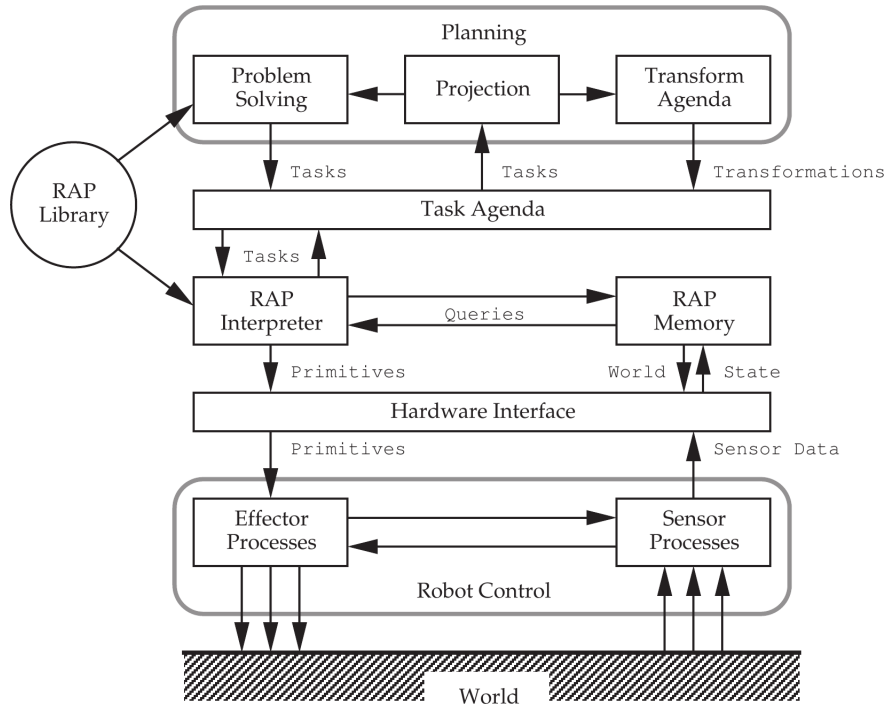


Figure A.4: A complete three-layered robot control system as envisaged by James Firby. The RAP system builds the middle layer (source: [Firby 89]).

task. This is information about a task's goal and a number of methods to execute the task in different situations. All RAPs are stored in the RAP library, i.e. the library contains information about all tasks the robot is able to carry out. Due to the modularity of the approach, it is easy to add further RAPs to enable the robot to execute new tasks. The RAP memory contains a world model created from sensor data. The execution of tasks is coordinated by the RAP interpreter, which selects tasks from the task agenda with regard to temporal deadlines and ordering constraints. Depending on the situation, an appropriate method is selected from the RAP that corresponds to the task. In case the selected method consists of primitive commands, it is directly executed by the low-level control system. Otherwise, the method contains subtasks, which are associated with corresponding RAPs and sent to the task agenda for later execution. That way, hierarchical plans with sequences of tasks can be created and executed.

James Firby explains that he sees the RAP system as an interface between what he calls the processes of action and deliberation. He believes that a number of tightly coupled components with access to fast sensor feedback are required for executing tasks. Although in [Firby 89] a three-level control system is described, only the middle layer was realised in the beginning. The RAP system has been combined with a modular, skill-based execution system consisting of behaviour control processes to form the Animate Agent Architecture (see [Firby 94]). In [Firby 95], the later addition of a spatial planning system is described. It shall be pointed out that the low-level control is realised in a different way than the definition of sequences of tasks. Hence, there is a breach between the lowest and the middle layer concerning the development of such a system.

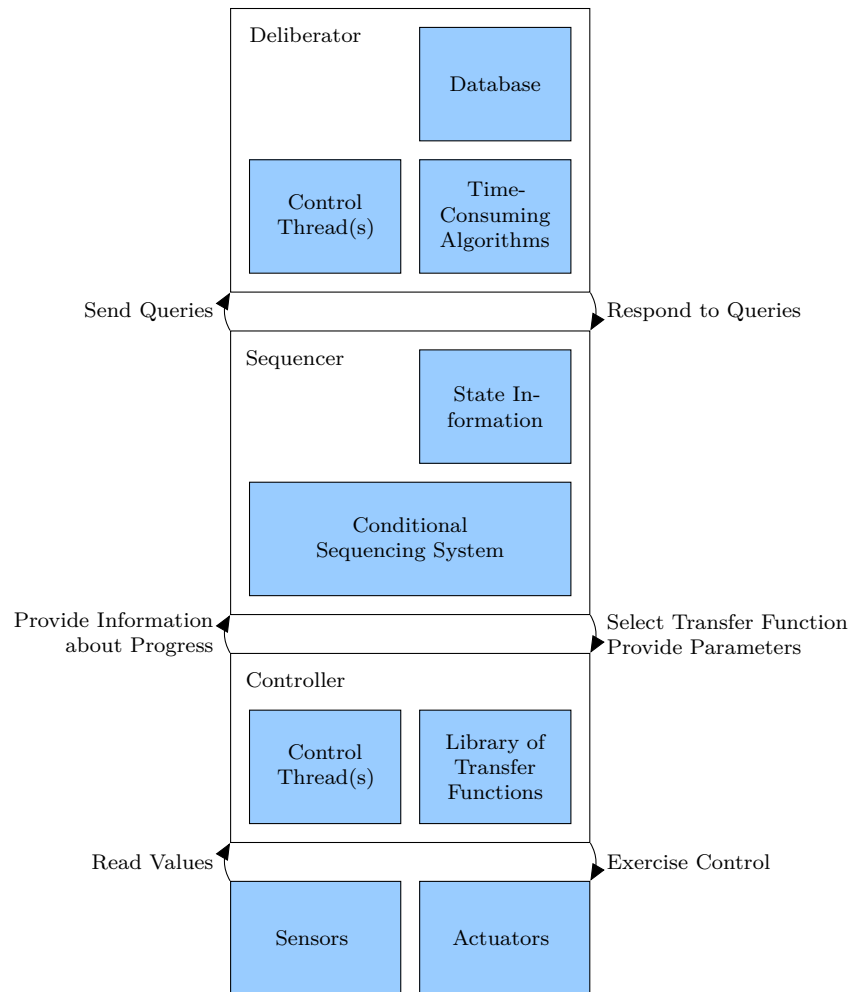


Figure A.5: A high-level view on ATLANTIS with its three main components controller, sequencer, and deliberator as well as their interaction. The information for this diagram has been taken from [Gat 92] and [Gat 98].

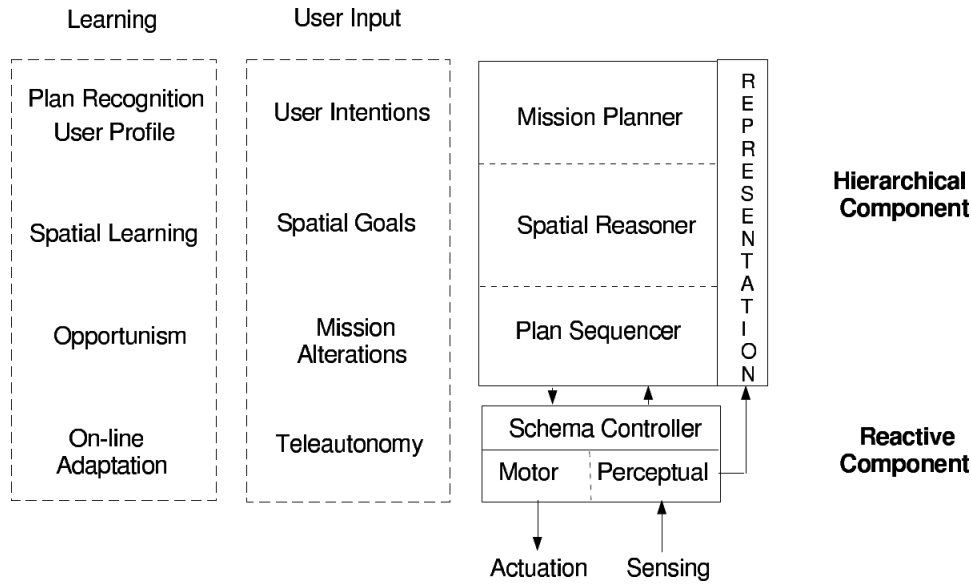


Figure A.6: The structure of AuRA with its deliberative, sequencing, and reactive components (right column). The column in the middle depicts the different types of user input, while the left one provides information about the types of learning on the different layers (source: [Arkin 97]).

A similar architecture is ATLANTIS², which is described by Erann Gat in [Gat 92] and [Gat 98]. Figure A.5 presents a high-level view on ATLANTIS. At the lowest level, the controller is responsible for dealing with the robot's sensors and actuators. Gat explains that this level is mostly reactive and consists of interconnected modules. One or more threads compute hand-crafted transfer functions that tightly couple sensors to actuators, i.e. the controller is designed in a reactive fashion. The second layer of ATLANTIS is called sequencer and resembles the RAP system. One of its tasks is to control sequences of primitive activities by activating and deactivating sets of modules in the controller and by monitoring the outcomes of these module activations. Furthermore, it initiates and terminates deliberative computations in the highest layer, the deliberator, which employs classical artificial intelligence algorithms. Gat argues that robot control architectures should be heterogeneous to support different computational mechanisms.

Another similar architecture is 3T (see [Bonasso 97]). It originated from the same research history and also features a modular low-level layer, a RAP-based sequencer, and a high-level planner. The authors of [Bonasso 97] explain that in contrast to ATLANTIS, the planner of 3T does not have to be specifically called by the sequencer. Again, the execution of single tasks and their sequencing are realised with different techniques in different layers, resulting in a breach within the control system.

AuRA is a further example (see [Arkin 97]). Although AuRA's structure differs slightly from the standard three-layered approach, it features a collection of low-level components (so-called schemas), a spatial planner, and a sequencer between the two (compare Fig. A.6). Different layers of the architecture take different types of user commands as input and employ different types of learning strategies.

²ATLANTIS: A Three-Layer Architecture for Navigating Through Intricate Situations

There are several other architectures which consist of a reactive component for low-level control, a deliberative component for high-level planning, and a component in between that serves as an interface between the two and performs the sequencing of tasks. A characteristic of these approaches is that different techniques are used for realising the different layers, resulting in certain breaches between them.

B. MCA and FINROC

The concepts presented in this dissertation have been implemented using the software frameworks MCA2 and FINROC. Therefore, the aspects of the two frameworks relevant to the work at hand will be described in the following sections.

B.1 MCA

MCA2¹ is a C++-based software framework for the development of robot control systems. It originates from the Research Center for Information Technology (“Forschungszentrum Informatik” or simply FZI in German) at the Karlsruhe Institute of Technology (KIT). An early version of MCA2 is described in [Scholl 01]. Since 2003, a branch of MCA2 has been developed in the Robotics Research Lab² of the Department of Computer Science³ at the University of Kaiserslautern⁴. This branch is now called MCA2-KL⁵.

As its name suggests, MCA2 strongly fosters the development of modular robot control systems. Its basic units are modules, which contain the actual functionality of a system and can be interconnected to build large networks. Their interfaces feature four types of ports over which real numbers can be transmitted: sensor input and output ports as well as controller input and output ports. While the former are used for transferring sensor data through a network, the latter are meant for sending and receiving, respectively, actuator commands. For the exchange of complex data between modules, blackboards can be used. Several modules can be combined to composite components (so-called groups), which can be connected in the same way as modules. In this manner, hierarchical systems can be built. The first implementation of the behaviour architecture iB2C has been realised in MCA2-KL.

There are two graphical tools available: the MCAGUI and the MCABrowser. The MCAGUI is mainly used for sending commands to a robot and for visualising the data of its sensors.

¹MCA2: Modular Controller Architecture

²website: <http://rrlab.cs.uni-kl.de/>

³website: <http://cs.uni-kl.de/>

⁴website: <http://www.uni-kl.de/>

⁵MCA2-KL: Modular Controller Architecture Version 2 - Kaiserslautern Branch; website: <http://rrlib.cs.uni-kl.de/mca-kl/>

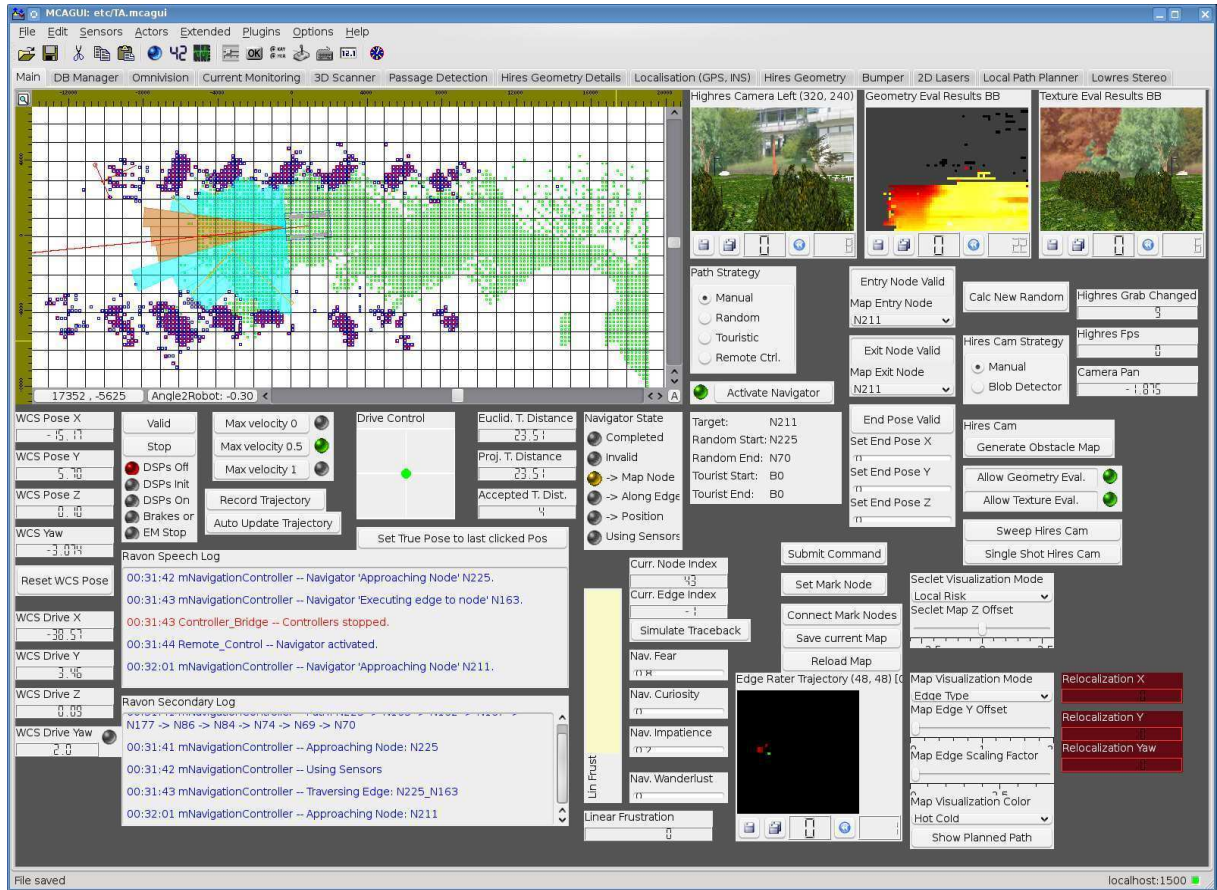


Figure B.1: A window of the GUI that is used for controlling the simulation of RAVON (source: [Armbrust 10b]).

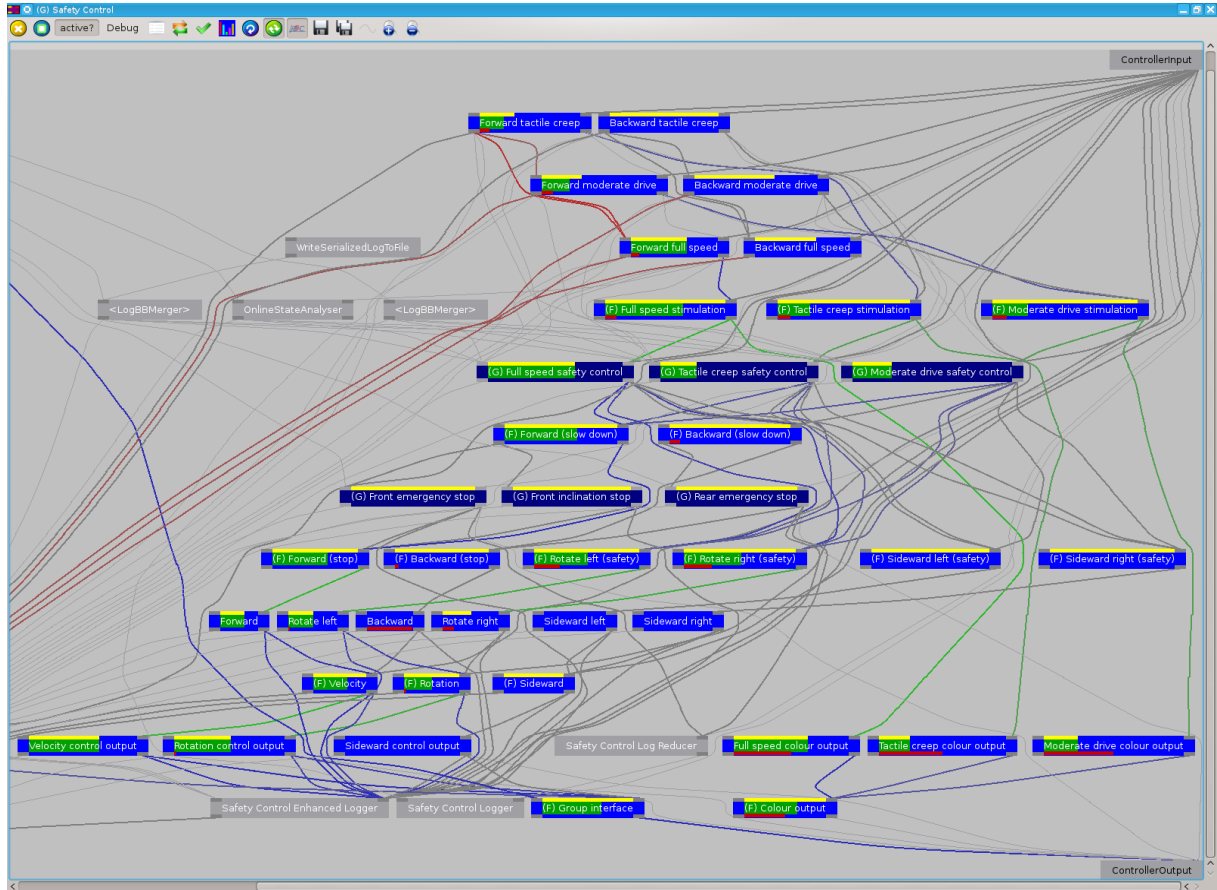


Figure B.2: The MCABrowser visualisation of a part of the behaviour-based anti-collision system of RAVON.

A typical application of the MCAGUI is the control of a robot during its operation. As the MCAGUI is freely configurable, it can be used to control various types of robots. For example, Fig. B.1 depicts the GUI used to control the simulation of the autonomous off-road robot RAVON.

The MCABrowser is used to gain insight into the state and structure of a control system as well as into the data flow within the control network. Typically, it is used for configuring the control system of a robot by setting parameters before the robot is sent on a mission or for analysing the system during the development and implementation phases. Figure B.2 depicts the MCABrowser visualisation of a part of the behaviour-based anti-collision system of RAVON. The MCABrowser features special support for visualising iB2C networks, with which a user can very quickly get an insight into the state of the network: While standard behaviours and fusion behaviours are depicted in light blue, behaviour-based groups are depicted in dark blue. CBS nodes are depicted in orange. Horizontal bars inside a behaviour or group visualise the component's activation (yellow), activity (green), and target rating (red). Edges represent data connections between components. Their colour provides information about the type of the connection (green: stimulation; red: inhibition; blue: activity transfer).

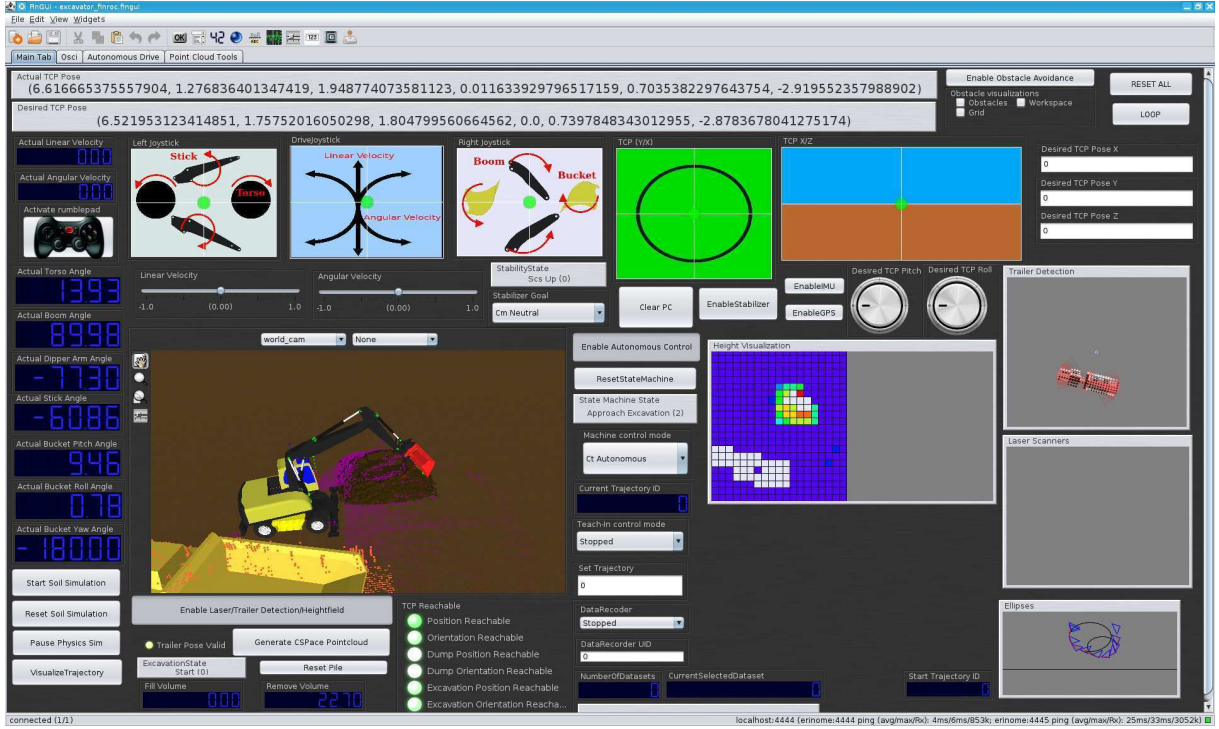


Figure B.3: A window of the GUI that is used for controlling the simulation of THOR.

B.2 FINROC

With a number of weak points of MCA2 (e.g. its monolithic kernel) in mind, researchers of the Robotics Research Lab at the University of Kaiserslautern have started the development of a downward compatible successor: FINROC⁶. There is a C++ and a Java implementation of FINROC. Besides its highly modular framework core, FINROC's efficient implementation of intra-process communication and its support for the construction of components during runtime are main advantages over MCA2. In addition, FINROC does not only support the MCA2 type of component with sensor and controller ports, but also other types. The iB2C has also been implemented in FINROC. Extensive information about FINROC can be found in [Reichardt 13].

FINROC features two graphical tools that are similar to the MCAGUI and the MCABrowser: the FINGUI and FINSTRUCT. Figure B.3 depicts the FINGUI-based interface for controlling the simulation of the autonomous bucket excavator THOR.

FINSTRUCT is the FINROC-analogue of the MCABrowser. It can also be used to visualise the structure of a robot control system. Moreover, it allows for graphically assembling control networks at runtime. Like the MCABrowser, it features a special visualisation of iB2C networks. Figure B.4 depicts the FINSTRUCT visualisation of a part of the behaviour-based control system of THOR. The most striking difference to the visualisation of behaviour networks in the MCABrowser is the colour coding: Standard behaviours are visualised with grey octagons, fusion behaviours with octagons in three different types of blue (depending on the type of fusion), and CBS nodes with orange hexagons. Behavioural groups are

⁶FINROC: Framework for Intelligent Robot Control; website: <http://www.finroc.org/>

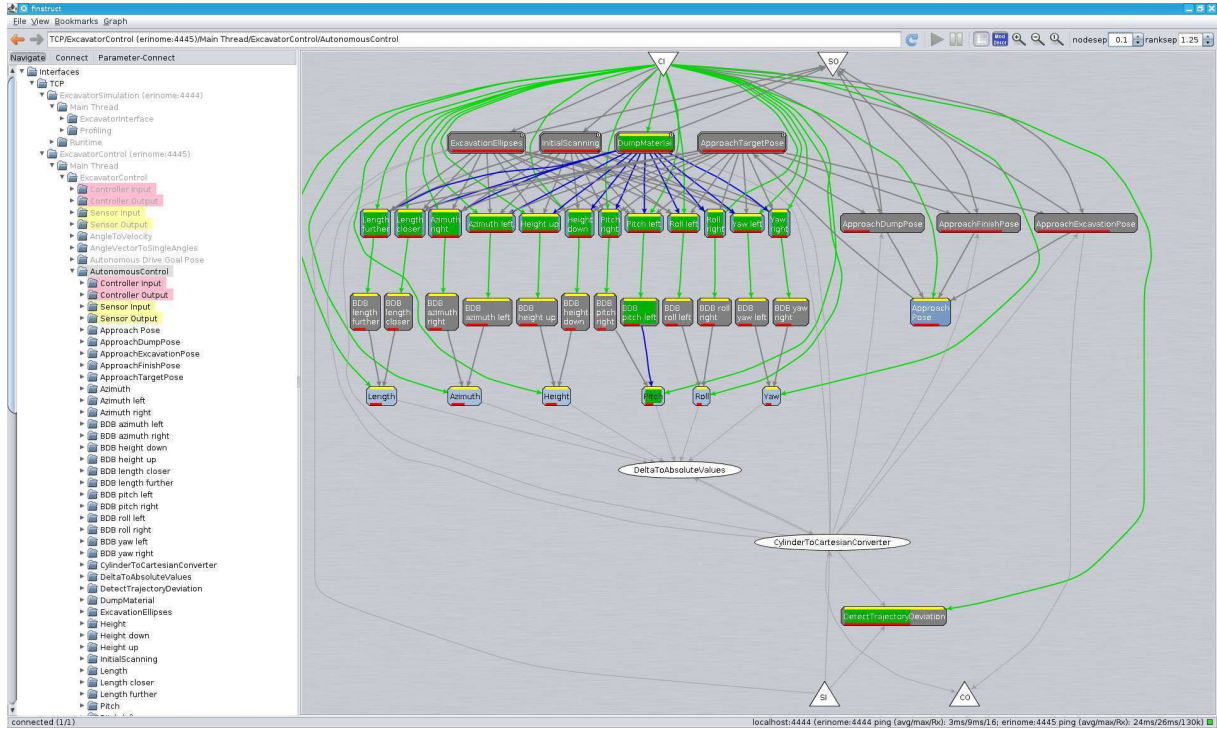


Figure B.4: The FINSTRUCT visualisation of a part of the behaviour-based control system of THOR.

represented by grey octagons with a double line as boundary. As in the MCABrowser, horizontal bars indicate a behaviour's activation (yellow), activity (green), and target rating (red). The colour of edges between components provides information about the type of the connection (green: stimulation; red: inhibition; blue: activity transfer). Non-behaviour modules are depicted with ellipses (single line: non-behaviour module; double line: non-behaviour group). In the context of the work described in this dissertation, FINSTRUCT has been extended with several widgets, e.g. for designing Moore machines (see Sec. 3.2.4) or query graphs (see Sec. 4.3.2.2).

In order to use FINROC programs (e.g. the graphical tools) together with MCA2 programs, there is a special mode for FINROC, called MCA2-legacy-mode. Using this mode, it is, for example, possible to run a robot control system implemented in MCA2, analyse it with FINSTRUCT, and control the robot using the FINGUI.

Bibliography

- [Alami 98] R. Alami, R. Chatila, S. Fleury, M. Ghallab, F. Ingrand, “An Architecture for Autonomy”, *International Journal of Robotics Research*, vol. 17, no. 4, pp. 315–337, April 1998.
- [Albus 87] J. S. Albus, H. G. McCain, R. Lumia, “NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) (NBS Technical Note 1235)”, National Bureau of Standards (NBS), Tech. Rep., July 1987.
- [Albus 89a] J. S. Albus, R. Lumia, J. Fiala, A. Wavering, “NASREM: The NASA/NBS Standard Reference Model for Telerobot Control System Architecture”, in *Proceedings of the 20th International Symposium on Industrial Robots*. Tokyo, Japan, October 4–6 1989.
- [Albus 89b] J. S. Albus, H. G. McCain, R. Lumia, “NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) (NBS Technical Note 1235, 1989 Edition)”, National Bureau of Standards (NBS), Tech. Rep., April 1989. supersedes NBS Technical Note 1235.
- [Alur 90] R. Alur, D. Dill, “Automata For Modeling Real-Time Systems”, in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science (LNCS), M. S. Paterson, Ed., Springer Berlin Heidelberg, 1990, vol. 443, pp. 322–335.
- [Alur 93] R. Alur, C. Courcoubetis, D. Dill, “Model-Checking in Dense Real-Time”, *Information and Computation*, vol. 104, no. 1, pp. 2–34, May 1993.
- [Arbib 81] M. A. Arbib, A. J. Kfoury, R. N. Moll, *A basis for theoretical computer science*, ser. Texts and monographs in computer science, Springer-Verlag, 1981. ISBN-10: 0-387-90573-1.
- [Arkin 87] R. C. Arkin, “Towards Cosmopolitan Robots: Intelligent Navigation in Extended Man-made Environments”, Dissertation, Graduate School of the University of Massachusetts, September 1987.
- [Arkin 89] R. C. Arkin, “Motor schema-based mobile robot navigation”, *International Journal of Robotics Research*, vol. 8, no. 4, pp. 92–112, August 1989. this publication is available at <http://dx.doi.org/10.1177/027836498900800406>.
- [Arkin 94] R. C. Arkin, D. MacKenzie, “Temporal Coordination of Perceptual Algorithms for Mobile Robot Navigation”, *IEEE Transactions on Robotics and Automation*, vol. 10, no. 3, pp. 276–286, June 1994.

- [Arkin 97] R. C. Arkin, T. Balch, “AuRA: Principles and Practice in Review”, *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2–3, pp. 175–189, 1997.
- [Arkin 98] R. Arkin, *Behaviour-Based Robotics*, MIT Press, 1998. ISBN-10: 0-262-01165-4; ISBN-13: 978-0-262-01165-5.
- [Armbrust 07] **C. Armbrust**, J. Koch, U. Stocker, K. Berns, “Mobile Robot Navigation Support in Living Environments”, in *20. Fachgespräch Autonome Mobile Systeme (AMS)*. Kaiserslautern, Germany: Springer-Verlag, October 2007, pp. 341–346.
- [Armbrust 09a] **C. Armbrust**, T. Braun, T. Föhst, M. Proetzsch, A. Renner, B.-H. Schäfer, K. Berns, “RAVON — The Robust Autonomous Vehicle for Off-road Navigation”, in *Proceedings of the IARP International Workshop on Robotics for Risky Interventions and Environmental Surveillance 2009 (RISE 2009)*, IARP. Brussels, Belgium, January 12–14 2009.
- [Armbrust 09b] **C. Armbrust**, B.-H. Schäfer, K. Berns, “Using Passages to Support Off-road Robot Navigation”, in *Proceedings of the 6th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2009)*, J. Filipe, J. Andrade-Cetto, J.-L. Ferrier, Eds., Institute for Systems and Technologies of Information, Control and Communication (INSTICC). Milan, Italy, July 2–5 2009, pp. 189–194.
- [Armbrust 10a] **C. Armbrust**, T. Braun, T. Föhst, M. Proetzsch, A. Renner, B.-H. Schäfer, K. Berns, “RAVON – The Robust Autonomous Vehicle for Off-road Navigation”, in *Using robots in hazardous environments: Landmine detection, de-mining and other applications*, Y. Baudoin, M. K. Habib, Eds., Woodhead Publishing Limited, 2010, ch. 15. ISBN: 1 84569 786 3; ISBN-13: 978 1 84569 786 0.
- [Armbrust 10b] **C. Armbrust**, M. Proetzsch, B.-H. Schäfer, K. Berns, “A Behaviour-based Integration of Fully Autonomous, Semi-autonomous and Tele-operated Control Modes for an Off-road Robot”, in *Proceedings of the 2nd IFAC Symposium on Telematics Applications*, IFAC. Politehnica University, Timisoara, Romania, October 5–8 2010. invited paper.
- [Armbrust 11a] **C. Armbrust**, L. Kiebusch, K. Berns, “Using Behaviour Activity Sequences for Motion Generation and Situation Recognition”, in *Proceedings of the 8th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2011)*, J.-L. Ferrier, A. Bernard, O. Gusikhin, K. Madani, Eds., Institute for Systems and Technologies of Information, Control and Communication (INSTICC). Noordwijkerhout, The Netherlands: SciTePress - Science and Technology Publications, July 28–31 2011, pp. 120–127.
- [Armbrust 11b] **C. Armbrust**, S. A. Mehdi, M. Reichardt, J. Koch, K. Berns, “Using an Autonomous Robot to Maintain Privacy in Assistive Environments”, *Security and Communications Networks: Special Issue on Privacy and Security in Pervasive e-Health and Assistive Environments*, vol. 4, no. 11, pp. 1275–1293, November 2011.
- [Armbrust 11c] **C. Armbrust**, M. Proetzsch, K. Berns, “Behaviour-Based Off-Road Robot Navigation”, *KI - Künstliche Intelligenz*, vol. 25, no. 2, pp. 155–160, May 2011. this publication is available at <http://dx.doi.org/10.1007/s13218-011-0090-2>.

- [Armbrust 12a] **C. Armbrust**, L. Kiekbusch, T. Ropertz, K. Berns, “Verification of Behaviour Networks Using Finite-State Automata”, in *KI 2012: Advances in Artificial Intelligence*, B. Glimm, A. Krüger, Eds. Saarbrücken, Germany: Springer, September 24–27 2012.
- [Armbrust 12b] **C. Armbrust**, D. Schmidt, K. Berns, “Generating Behaviour Networks from Finite-State Machines”, in *Proceedings of the 7th German Conference on Robotics (ROBOTIK 2012)*. May 22–25 2012.
- [Armbrust 13a] **C. Armbrust**, L. Kiekbusch, T. Ropertz, K. Berns, “Quantitative Aspects of Behaviour Network Verification”, in *Proceedings of the 26th Canadian Conference on Artificial Intelligence*, ser. Lecture Notes in Computer Science (LNCS), O. Zaiane, S. Zilles, Eds., vol. 7884. Regina, Saskatchewan, Canada: Springer, May 28–31 2013.
- [Armbrust 13b] **C. Armbrust**, L. Kiekbusch, T. Ropertz, K. Berns, “Tool-Assisted Verification of Behaviour Networks”, in *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA 2013)*. Karlsruhe, Germany, May 6–10 2013.
- [Behrmann 02] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, W. Yi, “UPPAAL Implementation Secrets”, in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science, W. Damm, E.-R. Olderog, Eds., Springer Berlin Heidelberg, 2002, vol. 2469, pp. 3–22.
- [Behrmann 04] G. Behrmann, A. David, K. G. Larsen, “A Tutorial on UPPAAL”, in *Formal Methods for the Design of Real-Time Systems*, ser. LNCS, M. Bernardo, F. Corradini, Eds., vol. 3185. Springer Berlin / Heidelberg, 2004, pp. 200–236. ISBN: 978-3-540-23068-7; this publication is available at http://dx.doi.org/10.1007/978-3-540-30080-9_7.
- [Behrmann 06] G. Behrmann, A. David, K. G. Larsen, “A Tutorial on UPPAAL 4.0”, November 28 2006. Revised and extended version of [Behrmann 04].
- [Ben-Ari 01] M. Ben-Ari, *Mathematical logic for computer science*, 2nd rev. ed., Springer-Verlag, 2001. ISBN-10: 1-85233-319-7.
- [Bérard 01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, P. McKenzie, *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer-Verlag Berlin Heidelberg GmbH, 2001. ISBN-13: 978-3-540-41523-7; this publication is available at <http://dx.doi.org/10.1007/978-3-662-04558-9>.
- [Beyer 04] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar, “An Eclipse Plug-in for Model Checking”, *12th IEEE International Conference on Program Comprehension*, no. 251 – 255, p. 251, June 24–26 2004.
- [Blom 05] J. Blom, A. Hessel, B. Jonsson, P. Pettersson, “Specifying and Generating Test Cases Using Observer Automata”, in *Formal Approaches to Software Testing*, ser. Lecture Notes in Computer Science (LNCS), J. Grabowski, B. Nielsen, Eds., vol. 3395. Springer-Verlag, 2005, pp. 125–139. Revised Selected Papers of the 4th International Workshop, FATES 2004.

- [Boehm 81] B. W. Boehm, *Software engineering economics*, ser. Prentice-Hall Advances in Computing Science and Technology Series, R. T. Yeh, Ed., Englewood Cliffs, N. J. 07632: Prentice-Hall, Inc., 1981. ISBN-10: 0-13-822122-7.
- [Bohren 11] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mösenlechner, W. Meeussen, S. Holzer, “Towards Autonomous Robotic Butlers: Lessons Learned with the PR2”, in *Proceedings of the 2011 IEEE International Conference on Robotics and Automation (ICRA 2011)*. Shanghai, China, May 9–13 2011, pp. 5568–5575.
- [Bonasso 97] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, M. G. Slack, “Experiences with an architecture for intelligent, reactive agents”, *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2, pp. 237–256, 1997.
- [Booch 05] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, 2nd ed., ser. Object Technology Series, Addison-Wesley Professional, 2005. ISBN-10: 0-321-26797-4.
- [Braun 09] T. Braun, *Cost-Efficient Global Robot Navigation in Rugged Off-Road Terrain*, ser. RRLab Dissertations, Verlag Dr. Hut, 2009. ISBN: 978-3-86853-135-0.
- [Bronštejn 08] I. N. Bronštejn, K. A. Semendjajew, *Taschenbuch der Mathematik für Ingenieure und Studenten*, 7th ed., Wissenschaftlicher Verlag Harri Deutsch GmbH, 2008. ISBN: 978-3-8171-2007-9.
- [Brooks 86] R. A. Brooks, “A Robust Layered Control System for a Mobile Robot”, *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, pp. 14–23, April 1986.
- [Brooks 89a] R. A. Brooks, “A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network”, in *Proceedings of the 1989 IEEE International Conference on Robotics and Automation (ICRA 1989)*. Scottsdale, AZ, USA, May 14–19 1989, pp. 692–696.
- [Brooks 89b] R. A. Brooks, “A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network”, *Neural Computation*, vol. 1, no. 2, pp. 253–262, 1989.
- [Brooks 90] R. A. Brooks, “Elephants Don’t Play Chess”, *Robotics and Autonomous Systems*, vol. 6, no. 1–2, pp. 3–15, 1990.
- [Brooks 91a] R. A. Brooks, “Intelligence Without Reason”, Massachusetts Institute of Technology (MIT), Artificial Intelligence Laboratory, A.I. Memo No. 1293, 1991.
- [Brooks 91b] R. A. Brooks, “New Approaches to Robotics”, Artificial Intelligence Laboratory, MIT, Tech. Rep., 1991.
- [Buxton 70] J. N. Buxton, B. Randell, Eds., *Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee*. Brussels, Belgium: NATO Science Committee; available from Scientific Affairs Division, NATO, Brussels 39, Belgium, April 1970.
- [Clarke 08] E. M. Clarke, “The Birth of Model Checking”, in *25 Years of Model Checking*, ser. Lecture Notes in Computer Science (LNCS), O. Grumberg, H. Veith, Eds., Springer Berlin Heidelberg, 2008, vol. 5000, pp. 1–26.

- [Clarke 82] E. M. Clarke, E. A. Emerson, “Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic”, in *Logics of Programs - Workshop*, ser. Lecture Notes in Computer Science (LNCS), D. Kozen, Ed., vol. 131. Springer Berlin Heidelberg, 1982, pp. 52–71.
- [Clarke 99] E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, MIT Press, 1999. ISBN-10: 0-262-03270-8; ISBN-13: 978-0262032704.
- [Dahl 05] T. S. Dahl, C. Giraud-Carrier, “Incremental Development of Adaptive Behaviors using Trees of Self-Contained Solutions”, *Adaptive Behaviour*, vol. 13, no. 3, pp. 243–260, 2005.
- [Dwyer 98] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, “Property Specification Patterns for Finite-State Verification”, in *Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP '98)*. Clearwater Beach, Florida, USA: ACM, March 04–05 1998, pp. 7–15.
- [Dwyer 99] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, “Patterns in Property Specifications for Finite-State Verification”, in *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. Los Angeles, California, USA: ACM, May 16–22 1999, pp. 411–420.
- [Firby 89] R. J. Firby, “Adaptive Execution in Complex Dynamic Worlds”, Dissertation, Graduate School of Yale University, New Haven, Connecticut, USA, May 1989. Technical Report YALEU/CSD/RR #672.
- [Firby 94] R. J. Firby, “Task Networks for Controlling Continuous Processes”, in *Proceedings of the Second International Conference on AI Planning Systems*. June 1994, pp. 49–54.
- [Firby 95] R. J. Firby, “Lessons Learned from the Animate Agent Project (so far)”, in *Lessons Learned from Implemented Software Architectures for Physical Agents: Papers from the 1995 Spring Symposium*, H. Hexmoor, D. Kortenkamp, Eds. Association for the Advancement of Artificial Intelligence, Menlo Park, California, March 1995, pp. 92–96.
- [Floyd 67] R. W. Floyd, “Assigning Meanings to Programs”, in *Mathematical Aspects of Computer Science*, ser. Proceedings of Symposia in Applied Mathematics, J. T. Schwartz, Ed., vol. 19, American Mathematical Society. Providence, Rhode Island, USA, 1967, pp. 19–32.
- [Garousi 11] V. Garousi, “Experience in Developing a Robot Control Software”, *Computer and Information Science*, vol. 4, no. 1, pp. 3–13, January 2011.
- [Gat 92] E. Gat, “Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Mobile Robots”, in *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI)*. 1992.
- [Gat 93] E. Gat, “On the Role of Stored Internal State in the Control of Autonomous Mobile Robots”, *AI Magazine*, vol. 14, no. 1, pp. 64–73, 1993.

- [Gat 94] E. Gat, G. Dorais, “Robot Navigation by Conditional Sequencing”, in *Proceedings of the 1994 International Conference on Robotics and Automation (ICRA 1994)*. San Diego, California, USA: IEEE Computer Society, May 1994, pp. 1293–1299.
- [Gat 96] E. Gat, “ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents”, in *Working Notes of the AAAI Fall Symposium on Plan Execution*. AAAI, 1996.
- [Gat 98] E. Gat, “Three-Layer Architectures”, in *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. P. Bonasso, R. Murphy, Eds., AAAI Press / The MIT Press, 1998, pp. 195–210. ISBN: 0-262-61137-6; title of the draft: On Three-layer Architectures.
- [Hexmoor 95] H. Hexmoor, D. Kortenkamp, “Issues on Building Software for Hardware Agents”, *The Knowledge Engineering Review*, vol. 10, no. 3, pp. 301–304, 1995.
- [Hirth 12] J. Hirth, *Towards Socially Interactive Robots – Designing an Emotion-based Architecture*, ser. RRLab Dissertations, Dr. Hut Verlag, 2012. ISBN-13: 978-3-8439-0631-9.
- [Hoare 69] C. A. R. Hoare, “An Axiomatic Basis for Computer Programming”, *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, October 1969.
- [Holt 99] A. Holt, “Formal verification with natural language specifications: guidelines, experiments and lessons so far.”, *South African Computer Journal*, no. 24, pp. 253–257, 1999.
- [IEEE 1012 04] “IEEE 1012-2004 - IEEE Standard for Software Verification and Validation”, 2004.
- [IEEE 1471 00] “IEEE 1471-2000 - IEEE Recommended Practice for Architectural Description of Software-Intensive Systems”, 2000.
- [ISO 42010 11] “ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description”, 2011.
- [ITU 11] ITU. *Specification and Description Language - Overview of SDL-2010 (Recommendation ITU-T Z.100)*. International Telecommunication Union - Telecommunication Standardization Sector (ITU-T). December 2011.
- [Jalote 05] P. Jalote, *An integrated approach to software engineering*, 3rd ed., ser. Texts in Computer Science, Springer, 2005. ISBN-10 (hardback): 0-387-20881-X; ISBN-10 (eBook): 0-387-28132-0; ISBN-13 (hardback): 978-0387-20881-7; ISBN-13 (eBook): 978-0387-28132-2.
- [Kajita 08] S. Kajita, B. Espiau, “Legged Robots”, in *Springer Handbook of Robotics*, B. Siciliano, O. Khatib, Eds., Springer Berlin Heidelberg, 2008, ch. 16, pp. 361–389.
- [Kiekbusch 14] L. Kiekbusch, **C. Armbrust**, K. Berns, “Formal Verification of Behaviour Networks Including Hardware Failures”, in *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Padova, Italy, July 15–19 2014.

- [Koch 08] J. Koch, **C. Armbrust**, K. Berns, “Small Service Robots for Assisted Living Environments”, in *VDI/VDE Fachtagung Robotik*. Munich, Germany, June 11–12 2008.
- [Konolige 97a] K. Konolige, “COLBERT: A Language for Reactive Control in Sapphira”, in *KI-97: Advances in Artificial Intelligence*, ser. Lecture Notes in Computer Science, G. Brewka, C. Habel, B. Nebel, Eds., Springer Berlin Heidelberg, 1997, vol. 1303, pp. 31–52.
- [Konolige 97b] K. Konolige, K. Myers, E. Ruspini, A. Saffiotti, “The Sapphira Architecture: A Design for Autonomy”, *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, vol. 9, no. 2–3, pp. 215–235, 1997.
- [Kortenkamp 08] D. Kortenkamp, R. Simmons, “Robotic Systems Architectures and Programming”, in *Springer Handbook of Robotics*, B. Siciliano, O. Khatib, Eds., Springer Berlin Heidelberg, 2008, ch. 8, pp. 187–206.
- [Kuo 11] C.-H. Kuo, Y.-S. Syu, T.-C. Tsai, T.-S. Chen, “An Embedded Robotic Wheelchair Control Architecture with Reactive Navigations”, in *2011 IEEE Conference on Automation Science and Engineering (CASE)*. Trieste, Italy, August 24–27 2011, pp. 810–815.
- [Langosz 13] M. Langosz, L. Quack, A. Dettmann, S. Bartsch, F. Kirchner, “A Behavior-Based Library for Locomotion Control of Kinetically Complex Robots”, in *Proceedings of the Sixteenth International Conference on Climbing and Walking Robots (CLAWAR 2013)*. Sydney, Australia, July 14–17 2013, pp. 495–502.
- [Larsen 99] K. G. Larsen, C. Weise, W. Yi, J. Pearson, “Clock Difference Diagrams”, *Nordic Journal of Computing*, vol. 6, no. 3, pp. 271–298, Fall 1999.
- [Ledermann 82] W. Ledermann, S. Vajda, Eds., *Analysis*, ser. Handbook of applicable mathematics, John Wiley & Sons Ltd., 1982, vol. 4. ISBN-10: 0-471-10141-9.
- [Liu 02] Y. Liu, “A Hoare-Style Proof System for Robot Programs”, in *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, R. Dechter, R. S. Sutton, Eds., Association for the Advancement of Artificial Intelligence (AAAI). Edmonton, Alberta, Canada: The AAAI Press, Menlo Park, California, July 28–August 1 2002, pp. 74–79.
- [Loetzsch 06] M. Loetzsch, M. Risler, M. Jüngel, “XABSL - A Pragmatic Approach to Behavior Engineering”, in *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006)*. Beijing, October 9–15 2006, pp. 5124–5129. this publication is available at <http://dx.doi.org/10.1109/IROS.2006.282605>.
- [Luksch 10] T. Luksch, *Human-like Control of Dynamically Walking Bipedal Robots*, ser. RRLab Dissertations, Verlag Dr. Hut, 2010. ISBN: 978-3-86853-607-2.
- [Maes 90] P. Maes, “Situated Agents Can Have Goals”, *Robotics and Autonomous Systems Special Issue: Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, vol. 6, no. 1–2, pp. 49–70, 1990.

- [Matarić 08] M. J. Matarić, F. Michaud, “Behaviour-Based Systems”, in *Springer Handbook of Robotics*, B. Siciliano, O. Khatib, Eds., Springer Berlin Heidelberg, 2008, ch. 38, pp. 891–910.
- [Matarić 97] M. J. Matarić, “Behavior-Based Control: Examples from Navigation, Learning, and Group Behavior”, *Journal of Experimental and Theoretical Artificial Intelligence - Special Issue on Software Architectures for Physical Agents*, vol. 9, no. 2–3, pp. 323–336, 1997.
- [McMillan 00] K. McMillan, “Overview of Verification”, in *Verification of Digital and Hybrid Systems*, ser. NATO ASI Series, M. K. Inan, R. P. Kurshan, Eds., Springer Berlin Heidelberg, 2000, vol. 170, pp. 3–13. ISBN-13: 978-3-642-64052-0.
- [Mehdi 09] S. A. Mehdi, **C. Armbrust**, J. Koch, K. Berns, “Methodology for Robot Mapping and Navigation in Assisted Living Environments”, in *PETRA '09: Proceedings of the 2nd International Conference on PErvasive Technologies Related to Assistive Environments*. Corfu, Greece: ACM, New York, NY, USA, June 9–13 2009. ISBN-13: 978-1-60558-409-6.
- [Meolic 01] R. Meolic, T. Kapus, Z. Brezonik, “CTL and ACTL patterns”, in *International Conference on Trends in Communications (EUROCON'2001)*, vol. 2. Bratislava, Slovakia, July 4–7 2001, pp. 540–543.
- [Nebel 12] M. E. Nebel, *Formale Grundlagen der Programmierung*, ser. Studienbücher Informatik, Springer Vieweg, 2012. ISBN-13 (print): 978-3-8348-1889-8; ISBN-13 (online): 978-3-8348-2296-3.
- [Newell 90] A. Newell, *Unified Theories of Cognition*, Harvard University Press, 1990. ISBN: 0-674-92099-6.
- [Nicolescu 00] M. N. Nicolescu, M. J. Matarić, “Extending Behavior-Based Systems Capabilities Using An Abstract Behavior Representation”, in *Working Notes of the AAAI Fall Symposium on Parallel Cognition*. North Falmouth, MA, November 3–5 2000, pp. 27–34.
- [Nicolescu 01a] M. N. Nicolescu, M. J. Matarić, “Experience-Based Learning of Task Representations from Human-Robot Interaction”, in *Proceedings of the 2001 IEEE International Symposium on Computational Intelligence in Robotics and Automation*. Banff, Alberta, Canada, July 29–August 1 2001, pp. 463–468.
- [Nicolescu 01b] M. N. Nicolescu, M. J. Matarić, “Experience-Based Representation Construction: Learning from Human and Robot Teachers”, in *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001)*. Maui, Hawaii, USA, October 29 – November 3 2001, pp. 740–745.
- [Nicolescu 02] M. N. Nicolescu, M. J. Matarić, “A Hierarchical Architecture for Behavior-Based Robots”, in *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*. Bologna, Italy, July 15–19 2002, pp. 227–233.

- [Nicolescu 03] M. N. Nicolescu, M. J. Matarić, “Linking Perception and Action in a Control Architecture for Human-Robot Domains”, in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS-36)*. IEEE Computer Society, January 6–9 2003.
- [Nicolescu 07] M. Nicolescu, O. C. Jenkins, A. Stanhope, “Fusing Robot Behaviours for Human-Level Tasks”, in *Proceedings of the IEEE 6th International Conference on Development and Learning (ICDL 2007)*. London, UK, July 11–13 2007, pp. 76–81.
- [Petri 62] C. A. Petri, “Kommunikation mit Automaten”, Dissertation, Technische Hochschule Darmstadt, 1962.
- [Pirjanian 99] P. Pirjanian, “Behaviour Coordination Mechanisms — State-of-the-art”, Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California, Tech. Rep. IRIS-99-375, October 1999.
- [Pluzhnikov 12] S. Pluzhnikov, D. Schmidt, J. Hirth, K. Berns, “Behavior-based Arm Control for an Autonomous Bucket Excavator”, in *Proceedings of the 2nd Commercial Vehicle Technology Symposium (CVT 2012)*, K. Berns, C. Schindler, K. Dreßler, B. Jörg, R. Kalmar, G. Zolynski, Eds. Kaiserslautern, Germany: Shaker Verlag, March 13–15 2012, pp. 251–261.
- [Pnueli 79] A. Pnueli, “The Temporal Semantics of Concurrent Programs”, in *Semantics of Concurrent Computation - Proceedings of the International Symposium*, ser. Lecture Notes in Computer Science (LNCS), G. Kahn, Ed., vol. 70. Evian, France: Springer Berlin Heidelberg, July 2–4 1979, pp. 1–20.
- [Priese 08] L. Priese, H. Wimmel, *Petri-Netze*, 2nd ed., Springer, 2008. ISBN-13: 978-3-540-76970-5.
- [Proetzsch 07] M. Proetzsch, K. Berns, T. Schuele, K. Schneider, “Formal Verification of Safety Behaviours of the Outdoor Robot RAVON”, in *Proceedings of the Fourth International Conference on Informatics in Control, Automation and Robotics (ICINCO 2007)*, J. Zaytoon, J.-L. Ferrier, J. Andrade-Cetto, J. Filipe, Eds. Angers, France: INSTICC Press, May 9–12 2007, pp. 157–164.
- [Proetzsch 10] M. Proetzsch, *Development Process for Complex Behavior-Based Robot Control Systems*, ser. RRLab Dissertations, Verlag Dr. Hut, 2010. ISBN: 978-3-86853-626-3.
- [Queille 82] J.-P. Queille, J. Sifakis, “Specification and Verification of Concurrent Systems in CESAR”, in *International Symposium on Programming - Proceedings of the 5th Colloquium*, ser. Lecture Notes in Computer Science (LNCS), M. Dezani-Ciancaglini, U. Montanari, Eds., vol. 137. London, UK: Springer-Verlag, 1982, pp. 337–351.
- [Reichardt 13] M. Reichardt, T. Föhst, K. Berns, “On Software Quality-motivated Design of a Real-time Framework for Complex Robot Control Systems”, *Electronic Communications of the EASST*, vol. 60: Software Quality and Maintainability 2013, August 2013. this publication is available at <http://journal.ub.tu-berlin.de/eceasst/article/view/855>.

- [Reisig 10] W. Reisig, *Petrinetze - Modellierungstechnik, Analysemethoden, Fallstudien*, 1st ed., ser. Leitfäden der Informatik, B. Becker, F. Mattern, H. Müller, W. Schäfer, D. Wagner, I. Wegener, Eds., Vieweg+Teubner Verlag, 2010. ISBN-13: 978-3-8348-1290-2.
- [Reisig 85] W. Reisig, *Petri Nets - An Introduction*, ser. EATCS Monographs on Theoretical Computer Science, W. Brauer, G. Rozenberg, A. Salomaa, Eds., Springer-Verlag, 1985, vol. 4. ISBN-10: 3-540-13723-8; translation of the German original edition: W. Reisig, *Petrinetze*.
- [Risler 08] M. Risler, O. von Stryk, “Formal Behavior Specification of Multi-Robot Systems Using Hierarchical State Machines in XABSL”, in *AAMAS’08 - Workshop on Formal Models and Methods for Multi-Robot Systems*. Estoril, Portugal, May 13 2008.
- [Rittmann 12] M. Rittmann, “Modelling of Complex Behaviour Nodes Using Finite-State Automata”, Bachelor’s thesis, Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2012. unpublished; supervised by Christopher Armbrust.
- [Rohr 12] S. Rohr, “Visualisation of Traces Originating from Model Checking”, Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, thesis of master’s project, 2012. unpublished; supervised by Christopher Armbrust.
- [Rohr 13] S. Rohr, “Generating Behaviour Networks from Finite-State Machines”, Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, thesis of master’s project in the area of guided research (angeleitete Forschung), 2013. unpublished; supervised by Christopher Armbrust.
- [Ropertz 11] T. Ropertz, “Modelling of Behaviour-Based Systems”, Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, thesis of master’s project, 2011. unpublished; supervised by Christopher Armbrust.
- [Ropertz 12] T. Ropertz, “Graphical Support for the Verification of Behaviour Networks”, Master’s thesis, Robotics Research Lab, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2012. unpublished; supervised by Christopher Armbrust and Lisa Kiekbusch.
- [Rosenblatt 97] J. K. Rosenblatt, “DAMN: A Distributed Architecture for Mobile Navigation”, *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2–3, pp. 339 – 360, April 1997.
- [Saffiotti 97] A. Saffiotti, “The uses of fuzzy logic in autonomous robot navigation”, *Soft Computing*, vol. 1, no. 4, pp. 180–197, 1997.
- [Sakarovitch 09] J. Sakarovitch, *Elements of automata theory*, Cambridge University Press, 2009. ISBN-13: 978-0-521-84425-3; English translation of “Éléments de théorie des automates”.

- [Schäfer 11] B.-H. Schäfer, *Control System Design Schemata and their Application in Off-road Robotics*, ser. RRLab Dissertations, Verlag Dr. Hut, 2011. ISBN: 978-3-86853-865-6.
- [Schäfer 13] B.-H. Schäfer, **C. Armbrust**, T. Föhst, K. Berns, “The Application of Design Schemata in Off-Road Robotics”, *Intelligent Transportation Systems Magazine, IEEE*, vol. 5, no. 1, pp. 4–27, Spring 2013.
- [Scherer 05] S. Scherer, F. Lerda, E. M. Clarke, “Model Checking of Robotic Control Systems”, in *Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS 2005)*. Munich, Germany, September 5–8 2005.
- [Schmidt 10] D. Schmidt, M. Proetzsch, K. Berns, “Simulation and Control of an Autonomous Bucket Excavator for Landscaping Tasks”, in *Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA 2010)*. Anchorage, Alaska, USA, May 3–8 2010, pp. 5108–5113.
- [Schneider 05] K. Schneider, T. Schuele, “Averest: Specification, Verification, and Implementation of Reactive Systems”, in *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD 2005)*, J. Desel, Y. Watanabe, Eds. Saint-Malo, France: IEEE Computer Society, June 7–9 2005.
- [Schneider 09] K. Schneider, “The Synchronous Programming Language Quartz”, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Internal Report 375, December 2009.
- [Scholl 01] K.-U. Scholl, J. Albiez, B. Gassmann, “MCA - An Expandable Modular Controller Architecture”, in *Proceedings of the 3rd Real-Time Linux Workshop*. Milano, Italy, November 26–29 2001.
- [Shields 89] M. W. Shields, *Serielle und parallele Automaten - Einführung in die Theorie*, VCH Verlagsgesellschaft mbH, 1989. ISBN-10: 3-527-27815-X; German translation of “An introduction to automata theory”.
- [Siciliano 08] B. Siciliano, O. Khatib, Eds., *Springer Handbook of Robotics*, Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-23957-4.
- [Sommerville 11] I. Sommerville, *Software engineering*, 9th, international ed., Boston: Pearson, 2011. ISBN-10: 0-13-705346-0; ISBN-13: 978-0-13-705346-9.
- [Sperschneider 91] V. Sperschneider, G. Antoniou, *Logic : a foundation for computer science*, ser. International Computer Science Series, Addison-Wesley, 1991. ISBN-10: 0-201-56514-5.
- [Sperschneider 96] V. Sperschneider, B. Hammer, *Theoretische Informatik : eine problemorientierte Einführung*, ser. Springer-Lehrbuch, Springer, 1996. ISBN-10: 3-540-60860-5.
- [Wagner 06] F. Wagner, R. Schmuki, T. Wagner, P. Wolstenholme, *Modeling Software with Finite State Machines: A Practical Approach*, Taylor & Francis Group, LLC, 2006. ISBN-13: 978-0-8493-8086-0.

- [Werger 00] B. B. Werger, “Ayllu: Distributed Port-Arbitrated Behavior-Based Control”, in *Proceedings of the 5th International Symposium on Distributed Autonomous Robotic Systems (DARS)*. Knoxville, Tennessee, USA: Springer Verlag, October 2000, pp. 25–34.
- [Winfield 09] A. F. T. Winfield, “Foraging Robots”, in *Encyclopedia of Complexity and Systems Science*, R. A. Meyers, Ed., New York: Springer, 2009, pp. 3682–3700. ISBN-13: 978-0-387-75888-6 (Print), 978-0-387-30440-3 (Online); this publication is available at: http://dx.doi.org/10.1007/978-0-387-30440-3_217.
- [Wongwirat 08] O. Wongwirat, T. Hanidthikul, N. Vuthikulvanich, “A Formal Approach in Robot Development Process using a UML Model”, in *Proceedings of the 10th International Conference on Control, Automation, Robotics and Vision (ICARCV 2008)*. Hanoi, Vietnam, December 17–20 2008.
- [Ziparo 06] V. A. Ziparo, L. Iocchi, “Petri Net Plans”, in *Proceedings of the Fourth International Workshop on Modeling of Objects, Components, and Agents (MOCA '06)*. Turku, Finland, 2006, pp. 267–290.
- [Ziparo 11] V. A. Ziparo, L. Iocchi, P. U. Lima, D. Nardi, P. F. Palamara, “Petri Net Plans - A framework for collaboration and coordination in multi-robot systems”, *Autonomous Agents and Multi-Agent Systems*, vol. 23, no. 3, pp. 344–383, 2011.
- [Zolynski 12] G. Zolynski, C. Schank, K. Berns, “Point Cloud Gathering for an Autonomous Bucket Excavator in Dynamic Surroundings”, in *Proceedings of the 2nd Commercial Vehicle Technology Symposium (CVT 2012)*, K. Berns, C. Schindler, K. Dreßler, B. Jörg, R. Kalmar, G. Zolynski, Eds. Kaiserslautern, Germany: Shaker Verlag, March 13–15 2012, pp. 262–271.

Index

- 3T, 20, 196
- abstract behaviour, *see* behaviour,
 - abstract
- acceptor, 9
- Ackermann steering geometry, 59
- Actions* port, 37
- activated, 26
- activation, 25
- ActivationCalculation, 108, 111
- active, 26
- Active* port, 37
- activity, 25
 - derived, 25
- activity function, 26
- activity vector, 25
- ActivityCalculation, 108, 112
- ActivLevel* port, 36
- APF, *see* artificial potential field
- arbitration
 - priority-based, 28, 33
- architecture, 189
 - behaviour-based, 21, 22
 - deliberative, 20, 190
 - hybrid, 20, 193
 - reactive, 20, 191
 - subsumption, 33
 - three-layer, 20, 193
 - USC behaviour, 36, 44
- artificial potential field, 192
- ARTOS, 30
- asynchronous_before, 150
- asynchronous_paired_before, 151
- asynchronous_requires_once, 151
- ATLANTIS, 20, 196
- AuRA, 21, 196
- Averest, 32
- AYLLU, 36
- basic behaviour, *see* behaviour, basic
- BD*, *see* Blockade Detector
- BDD, *see* binary decision diagram
- behaviour
 - abstract, 36, 44
 - basic, 78
 - feedback, 45
 - fusion, 26
 - input, 45
 - local coordination, 44
 - primitive, 36, 44
 - safety, 186
 - stimulated, 45
- behaviour activity sequence, *see* sequence,
 - behaviour activity
- behaviour group, 28
- behaviour signal, 25, 149
- behaviour-based architecture, *see*
 - architecture, behaviour-based
- behavioural group, *see* behaviour group
- binary decision diagram, 100
 - ordered, 99, 101
- binary decision tree, 99
- blockade, 67
- Blockade Detector*, 67, 71
- calculus, 95
- CBS, *see* conditional behaviour stimulator
- (CBS) *Cycle Init*, 65
- (CBS) *Dead End Detected*, 66, 71
- (CBS) *DED*, *see* (CBS) *Dead End Detected*
- (CBS) *Robot in Passage*, 66, 71
- (CBS) *RP*, *see* (CBS) *Robot in Passage*
- (CBS) *TA*, *see* (CBS) *Turn Around*
- (CBS) *Turn Around*, 61, 63
- CBSActivityCalculation, 124
- CBSCombineEnabling, 132
- CBSCombineOrdering, 132
- CBSCombinePermanent, 132
- CBSConditionsFulfilled, 124, 126, 132

- CBSConnectICAndFC, 124, 129, 132
- CBSEnablingFulfilled, 125, 132
- CBSInputChangedInterface, 124, 125, 130
- CBSOrderingFulfilled, 125
- CBSPermanentFulfilled, 125
- CBSTargetRatingCalculation, 124
- CDD, *see* clock difference diagram
- central pattern generator, 25
- clock difference diagram, 101
- Colbert, 78
- computation tree, 97
- Computation Tree Logic, 97
- condition
 - enabling, 46, 47
 - ordering, 46, 47
 - permanent, 46, 47
- conditional behaviour stimulator, 44, 45, 123
- conditional sequencing, 23
- conflicter link, 36
- conjunction, 82
- Continue* port, 37
- control value, 25
- correct
 - partially, *see* correctness, partial
 - totally, *see* correctness, total
- correctness
 - partial, 95
 - total, 95
- counterexample, 96, 148
- CPG, *see* central pattern generator
- CTL*, *see* Computation Tree Logic

- DAMN, 23
- DBM, *see* difference bounded matrix
- dead end, 67
- deduction, 95
- deductive reasoning, 95
- deliberative architecture, *see* architecture, deliberative
- derived activity, *see* activity, derived
- difference bounded matrix, 101
- disjunction, 82

- edge, 102
- Effects* port, 37

- embodiment, 21
- emergence, 21
- enabling condition, *see* condition, enabling
- enabling precondition, *see* precondition, enabling
- end user, 72
- Entering Passage*, 67, 71
- EP*, *see* *Entering Passage*
- ESL, *see* Execution Support Language
- eventually, 151
- Execution Support Language, 20
- existential property, 98
- exploration robot, 8
- exploration task, 8

- (F) *Combine Activities*, 65
- (F) *Combine Target Ratings*, 65
- (F) *Mediator*, 141
- (F) *Nav. DPA Interface*, *see* (F) *Navigator Direct Point Access Interface*
- (F) *Nav. PAO Interface*, *see* (F) *Navigator Point Access with Orientation Interface*
- (F) *Navigator Direct Point Access Interface*, 141
- (F) *Navigator Point Access with Orientation Interface*, 141
- (F) *Passage Driver Target*, 141
- (F) *PDT*, *see* (F) *Passage Driver Target*
- (F) *Point Access Mode*, 143
- (F) *Turn Around*, 65
- FBActivityCalculation, 117, 118, 120
- FBIBActivityCalculation, 118, 120
- FBTargetRatingCalculation, 121
- feedback behaviour, *see* behaviour, feedback
- feedback condition, 45
- feedback relation, 46
- feedback signal, 46
- feedback threshold, 46
- finite net, 17
- finite sequence, *see* sequence, finite
- finite-state automaton, 10
- finite-state machine, 10
- FINROC, 25, 202

- FSA, *see* finite-state automaton
- FSM, *see* finite-state machine
- fusion
 - maximum, 27
 - weighted average, 27
 - weighted sum, 28
- fusion behaviour, *see* behaviour, fusion
- fusion primitive, 28
- (G) Drive Control*, 139, 141
- (G) Local Path Planner*, 141
- (G) LPP*, *see* *(G) Local Path Planner*
- (G) Turn Around*, 59, 63
- globally, 151
- hybrid architecture, *see* architecture,
 - hybrid
- hypercube, 133
- iB2C, 25
- infinite sequence, *see* sequence, infinite
- Inhibit* port, 36
- inhibition, 25, 33
- inhibition vector, 25
- InhibitionInterface*, 108, 109
- input behaviour, *see* behaviour, input
- input condition, 45
- input relation, 46
- input signal, 46
- input threshold, 46
- input vector, 25
- intelligence, 21
- ITU-T, 19
- Kripke structure, 97
- level of competence, 33
- liveness property, *see* property, liveness
- local coordination behaviour, *see*
 - behaviour, local coordination
- location, 102
- main developer, 72
- maximum fusion, *see* fusion, maximum
- MCA2, 199
- MCA2-KL, 25, 199
- MCA2-legacy-mode, 203
- MCABrowser, 199
- MCAGUI, 199
- Mealy machine, 9
- model checking, 96, 97
 - explicit, 99
 - symbolic, 99
- Moore machine, 9
- narrow passage, *see* passage, narrow
- Narrow Passage Detector*, 61, 67, 68, 71
- NASA/NBS Standard Reference Model for
 - Telerobot Control System
 - Architecture, 190
- NASREM, *see* NASA/NBS Standard
 - Reference Model for Telerobot
 - Control System Architecture
- network abstract behaviour, 37
- New Passage*, 141
- NP*, *see* *New Passage*
- NPD*, *see* *Narrow Passage Detector*
- OA*, *see* *Orientation Activation*
- OBDD, *see* binary decision diagram,
 - ordered
- observer automaton, 147, 154
- OD*, *see* *Orientation Deactivation*
- ordering condition, *see* condition,
 - ordering
- ordering precondition, *see* precondition,
 - ordering
- Orientation Activation*, 59
- Orientation Deactivation*, 59
- output vector, 25
- P/T net, *see* place/transition net
- partial correctness, *see* correctness,
 - partial
- partially correct, *see* correctness, partial
- passage, 141
 - narrow, 61, 67
- Passage Driver*, 141
- passage entry, 67
- Passage Manager*, 141
- path formula, 103
- path quantifier, 97
- PD*, *see* *Passage Driver*
- permanent condition, *see* condition,
 - permanent
- permanent precondition, *see*
 - precondition, permanent

- Petri net, 16
- place/transition net, 18
 - capacity, 18
 - initial marking, 18
 - place, 18
 - transition, 18
 - weight, 18
- PM*, *see* *Passage Manager*
- point access, 139
 - direct, 139
 - with orientation, 139
- postcondition, 37, 95
- precedence, 143, 151
- precondition, 95
 - enabling, 38
 - ordering, 38
 - permanent, 38
 - sequential, 36
 - world, 36
- Precondition* port, 37
- predecessor link, 36
- primitive behaviour, *see* behaviour,
 - primitive
- principle, 26
- priority, 151
- property, 148
 - liveness, 103
 - reachability, 103
 - safety, 103
- property term, 149
- Quartz, 32
- query, 104, 138
- query edge, 153
- query graph, 153
- query language, 103
- query vertex, 153
- RAP, 20, 193
 - interpreter, 20, 194
 - library, 194
 - memory, 194
- RAVON, 30, 57, 59, 139
- reachability property, *see* property,
 - reachability
- reactive action package, *see* RAP
- reactive architecture, *see* architecture,
 - reactive
- relation symbol, 149
- requires_non-strict**, 151
- requires_strict**, 151
- reset input, 45
- robot control architecture, *see*
 - architecture
- robot control system, 189
- robotic wheelchair, 191
- ROS, 89
- rule, 95
- safety behaviour, *see* behaviour, safety
- safety property, *see* property, safety
- Same Passage*, 141
- SDL, 19
- SensoryInput* port, 37
- sequence
 - behaviour activity, 24
 - finite, 7
 - infinite, 7
 - task, 8
- sequencer, 20
- sequential precondition, *see* precondition,
 - sequential
- signal threshold, 149
- situatedness, 21
- SMACH, 89
- SMPA, 190
- SP*, *see* *Same Passage*
- SPA, 190
- Specification and Description Language,
 - see* SDL
- SPG, *see* spinal pattern generator
- spinal pattern generator, 25
- state automaton, 9
- state diagram, 10
- state explosion problem, 99
- state formula, 103
- state machine, 9
- stimulated behaviour, *see* behaviour,
 - stimulated
- stimulation, 25
- StimulationInterface**, 108, 109
- subsumption architecture, *see*
 - architecture, subsumption
- successor link, 36
- suppression, 33

synchronous_before, 150
synchronous_paired_before, 151
synchronous_requires_once, 151
system specialist, 74

target rating, 25
target rating function, 26
TargetRatingCalculation, 108, 113
task sequence, *see* sequence, task
TCTL, *see* Timed Computation Tree
 Logic
temporal logic, 97, 149
temporal operator, 98
temporal property, 149
THOR, 161
three-layer architecture, *see* architecture,
 three-layer
timed automaton, 102
Timed Computation Tree Logic, 103, 138
total correctness, *see* correctness, total
totally correct, *see* correctness, total
transducer, 9
transfer function, 26
Turn Around, 65
turning manoeuvre, 59, 61

UML, 14
 activity diagrams, 14
 state diagrams, 14
Unified Modeling Language, *see* UML
universal property, 98
UPPAAL, 101
USC behaviour architecture, *see*
 architecture, USC behaviour
UseBehaviour port, 36

validation, 94
verification, 94
verifier, 104, 138

weighted average fusion, *see* fusion,
 weighted average
weighted sum fusion, *see* fusion, weighted
 sum
witness, 96, 148
world precondition, *see* precondition,
 world

XABSL, 78

